FISH & RICHARDSON P.C.

Suite 1100
919 N. Market Street
P.O. Box 1114
Wilmington, Delaware
19899-1114

Telephone
302 652-5070

Facsimile
302 652-0607

Web Site
www.fr.com

Frederick P. Fish
1855-1930

W.K. Richardson
1859-1951

ATLANTA
AUSTIN
BOSTON
DALLAS
DELAWARE
NEW YORK
SAN DIEGO
SILICON VALLEY
TWIN CITIES
WASHINGTON, DC

December 20, 2006

**VIA E-FILING AND HAND DELIVERY**
**PUBLIC VERSION - December 29, 2006**

The Honorable Gregory M. Sleet
United States District Court
844 King Street, Room 4324
Wilmington, DE 19801

Re: Synopsys v. Magma Design Automation
USDC-D. Del. - C.A. No. 05-701-GMS

Dear Judge Sleet:

We write in response to Synopsys's request for permission to file four summary judgment motions. None is worth the time and effort required for full briefing.

## 1. "Octtools" Fails to Anticipate the '328 Patent

Synopsys claims that the '328 Patent is anticipated by U.C. Berkeley's "Octtools" software. But even assuming that Octtools was publicly available—a disputed factual issue because Synopsys has never deposed the original Octtools developers—Synopsys stipulated to a claim construction that renders Octtools irrelevant.

The parties stipulated that the term "common data model" in the '328 Patent means a model that "***does not require translation*** between the design tools." (Exh. A to the Second Amended Final Joint Claim Construction Charts filed Nov. 28, 2006 [D.I. 159], emphasis added.) Octtools requires translation during the design process, and thus fails to anticipate any claim of the '328 Patent.

As an example of translation in Octtools, "Oct" data objects do not include any object with a netlist portion,[1] (*see* Exh. A at 8-15), and a translator, BDNET, must be employed to translate netlist definition files to a format compatible with the Oct database. (*See* Exh. B at 27 ("We can create an OCT cell with 4 flip-flops ***using the netlist to Oct translator***, bdnet." (emphasis added).)) The Octtools hardware description translator operates similarly—because BDSYN cannot read from or write to the Oct database, users must manually enter the desired logic to the tool.[2] (*See* Exh. C at 2.) Similarly, when finalizing a circuit for fabrication, Octtools requires translation. (*See* Exh. D at 1 ("In order to submit Oct designs for fabrication, there

[1] Because Octtools objects cannot include an object with a netlist portion, Octtools also fails to anticipate the following element of claim 1: "certain ones of the objects including a netlist portion that represents a corresponding portion of the circuit." ('328 Patent at 9:8-10.)
[2] Although "common data model" appears in the preamble to Claim 1, it constitutes a "distinct definition" of the "model" in the body of the claims. *See Pitney-Bowes, Inc. v. Hewlett-Packard Co.*, 182 F.3d 1298, 1305 (Fed. Cir. 1999).

are a few housekeeping details that need to be addressed. One of these is conversion from an Oct database to a properly scaled CIF file."))

The '328 Patent advanced the art because it eliminated translations. *See, e.g.*, '328 Patent at 2:34-41. The '328 Patent successfully uses "common" data throughout the design process.[3] Octtools lacks that central feature.[4]

The '328 Patent also requires that each of the objects representing the circuit be "logically correlated to at least one other object." ('328 Patent at 9:11; Claim 1.) Octtools' data objects, however, need not be "logically correlated"—attachment of data objects is optional. (*See* Exh. E.) And even if attached, the correlation between objects need not be logical. For example, Octtools permits the attachment of a "text box" to objects in its database, even though a text box has no relevance to the logical operation of a circuit. (*See* Exh. F; Exh. A at 11-12.)

These are only some of the ways in which Octtools fails to anticipate the '328 Patent, as a matter of law. Moreover, there are genuine issues of material fact regarding its public availability. Also, because its the jury's role is to consider each party's evidence and to evaluate the scope and content of the prior art, this issue is inappropriate for resolution on summary judgment.

## 2. The Claims of the '116 and '093 Patents Are Sound

Synopsys relies on a single—and highly fact-specific—Federal Circuit opinion, *IPXL Holdings, LLC v. Amazon.com, Inc.*, 430 F.3d 1377 (Fed. Cir. 2005), in an effort to invalidate forty-six claims from two Magma patents on the ground that they are "mixed apparatus and method" claims. ***Every district court that has subsequently faced this issue has rejected the argument now being made by Synopsys***, including: *Toshiba Corp. v. Juniper Networks, Inc.*, 2006 U.S. Dist. LEXIS 44348 (D. Del. June 28, 2006); *Collegenet, Inc. v. XAP Corp.*, 442 F.Supp.2d 1036 (D. Ore. 2006); *Collaboration Props., Inc. v. Tandberg ASA,* 2006 U.S. Dist. LEXIS 42465 (N.D. Cal. June 23, 2006); and *Yodlee, Inc. v. Cashedge, Inc.*, 2006 U.S. Dist. LEXIS 86699 (N.D. Cal. November 29, 2006).

The *IPXL Holdings* case does not apply to these claims. Computer hardware and software claims are often drafted using functional language to describe the steps the computer takes when it executes. *Id.; see also Yodlee* at *11-12 ("A computer-readable storage device storing instructions that upon execution cause a processor to automatically access personal information ... by performing the steps

[3] Indeed, Synopsys does not even argue—much less demonstrate—that the Oct database is capable of using common data throughout the design process.
[4] *See* Exh. G ("The system includes tools for PLA and multiple-level logic synthesis, . . . ***Most tools are integrated*** with the Oct data manager and the VEM user interface.") (emphasis added).

comprising…"). *IPXL* involved a claim that covered not only a computer system but also the ***user using that system***. *See* 430 F.3d at 1384. Here, Magma's claims are all directed to the computer system itself, not to the user. Functional language to describe how that system operates is both permissible and commonplace in computer-related patents. There isn't a summary judgment issue here.

### 3. Synopsys Ignores The Parties' Agreement To Exchange Supplemental Claim Charts

Synopsys argues that Magma's infringement chart for the '116 Patent is insufficiently detailed. But this was one of the issues before the Court in the December 12, 2006 discovery hearing, and as instructed by the Court, the parties have met and conferred about supplementation of their respective infringement charts. Synopsys's request to file a summary judgment motion ignores the parties' agreement to exchange more detailed infringement charts.

Although Magma will serve its revised '116 chart on December 26, 2006, a brief summary of the chart is attached as Exhibit H, addressing claims 1 and 3-5, as highlighted in Synopsys's opening letter. Clearly, there is a logical, grounded basis for infringement here, and extensive summary judgment briefing would be a waste of time and paper.[5]

### 4. Synopsys's Motion for Summary Judgment on Counterclaims 1 - 6

Synopsys's request to move for summary judgment on Magma's six non-patent claims misrepresents Magma's allegations, the law, and the evidence.

Magma's non-patent claims—monopolization and attempted monopolization, trade libel, unfair competition, and tortious interference with business expectancies—are ***not*** based exclusively on Synopsys's assertion of the scan chain patents and its statements about this litigation. Rather, each of these claims is also based on Synopsys's false and disparaging comments and predatory pricing arrangements (*see, e.g.*, Magma's Second Amended Answer [D.I. 76] at ¶¶ 115, 116), with resulting harm to Magma and the public (*id.* at ¶¶ 117-119). Magma's antitrust allegations also reference this behavior. (*Id.* at ¶¶143, 148.)

5 Synopsys's argument that Magma identifies multiple products in its infringement contentions is a red herring. Synopsys sells and markets its products as a ***suite of tools that work together***, much as Microsoft sells Microsoft Office as a suite designed to work together. *See* http://www.synopsys.com/products/solutions/galaxy_platform.html ("The Galaxy™ Design Platform is a comprehensive solution for digital IC implementation."). Magma's contention that certain products work together to carry out infringing functions is not "defective as a matter of law," and it is notable that Synopsys cites no authority for that proposition.

Even if there were no deceptive intent in the procurement of the scan chain patents—and intent of course is a notoriously fact-driven issue depending on circumstantial evidence[6]—Magma's claims would still go to the jury. *See, LePage's Inc. v. 3M*, 324 F.3d 141 (3d Cir. 2003) (considering 3M's various activities as a whole and finding that the anticompetitive effect of 3M's bundled rebates and exclusive dealing arrangements were enough to maintain a Section 2 antitrust claim).[7]

There is also a material issue of fact regarding antitrust injury. While Magma's statements to investors have been upbeat,[8] Magma has ***specifically identified*** a variety of harmful effects of Synopsys's anticompetitive behavior, including

REDACTED

Further, Synopsys's *Noerr-Pennington* argument, which it did not even bother to raise in its unsuccessful motion to dismiss, collides directly with *Walker Process*. One cannot claim petitioning immunity for asserting fraudulently procured patents. *Walker Process v. Food Machinery*, 382 U.S. 172, 177 (1965).[9]

Finally, whether the scan chain patents would have issued over the omitted prior art does not determine materiality of the omitted prior art (an element of inequitable conduct). *See Digital Control Inc., v. Charles Machine Works,* 437 F.3d 1309, 1318 (Fed. Cir. 2006) ("a misstatement or omission may be material even if disclosure of that misstatement or omission would not have rendered the invention unpatentable"); *Bristol-Myers Squibb Co. v. Rhone-Poulenc Rorer, Inc.*, 326 F.3d 1226, 1237-38

[6]:

REDACTED

[7] Discovery is ongoing, with relevant depositions of the Synopsys Taiwan manager and corporate depositions of Synopsys on its statements to customers remaining to be taken, and associated documents yet to be received from Synopsys. It makes no sense to address summary judgment on such inherently complex factual issues when Synopsys has failed to provide such relevant discovery.
[8] The scheduled depositions of Roy Jewell and Rajeev Madhaven, whose public statements are noted by Synopsys, will provide context for those statements and provide further evidence of harm.
[9] Indeed, Synopsys's dedication of the scan chain patents to the public reinforces Magma's *Walker Process* antitrust claims.

(Fed. Cir. 2003) (withheld reference was material notwithstanding patent examiner's determination that it did not render the invention unpatentable); *Hoffmann-La Roche, Inc. v. Promega Corp.*, 323 F.3d 1354, 1367-68 (Fed. Cir. 2003) (misstatements regarding an "alternative" argument were material); *PerSeptive Biosystems, Inc. v. Pharmacia Biotech, Inc.*, 225 F.3d 1315, 1322 (Fed. Cir. 2000) (a patent may be valid and yet unenforceable due to inequitable conduct).

In view of the foregoing, we respectfully request that none of the four suggested summary judgment motions be approved for full briefing, and that these issues be reserved for trial.

Respectfully,

*/s/ William J. Marsden, Jr.*

William J. Marsden, Jr.

WJM/dob

cc: Karen Jacobs Louden, Esquire (via e-filing and hand delivery)
Valerie M. Wagner, Esquire (via e-mail and Federal Express)

80040483.doc

# A

# Octtools 5.2
# Part II: Reference

Electronics Research Laboratory
University of California at Berkeley

May 25, 1993

# Contents

1 **Getting Started** 1
- 1.1 Setting Up the UNIX Environment . . . 1
- 1.2 Documentation . . . 2
- 1.3 Installation . . . 2
- 1.4 How the Octtools Are Organized . . . 3
- 1.5 Making a Tool . . . 4

2 **The General Structure of Oct** 7
- 2.1 The OCT Objects . . . 8
- 2.2 The Oct Functions . . . 16
- 2.3 Useful packages . . . 29
- 2.4 Staging and Unstaging Facets . . . 29
- 2.5 Accessing OCT Objects . . . 31
- 2.6 Referring to OCT Objects . . . 31
- 2.7 Traversing the OCT Object Graph . . . 32
- 2.8 Instances and Masters . . . 33
- 2.9 Change Propagation and Dealing with Inconsistencies . . . 33
- 2.10 Monitoring Changes . . . 34
- 2.11 Performance . . . 36

3 **Octtools policies** 39
- 3.1 Oct Generic Policy . . . 39
- 3.2 Oct Physical Policy . . . 43
- 3.3 Oct Symbolic Policy . . . 44
- 3.4 Oct Schematic Policy . . . 47
- 3.5 Policy on OCT and Spice . . . 51
- 3.6 Symbolic/Schematic Properties . . . 52

4 **Guidelines for Inclusion of Tools and Packages** 55
- 4.1 Integrating tools with VOV . . . 58
- 4.2 Sample Programs . . . 61

5 **RPC: Remote Procedure Call** 65
- 5.1 How RPC Applications Work . . . 66
- 5.2 RPC Requirements . . . 66
- 5.3 Current RPC Applications . . . 66
- 5.4 Using RPC Applications . . . 67
- 5.5 Programming Interface . . . 68
- 5.6 Putting Together a Remote Application . . . 79
- 5.7 Example RPC Applications . . . 80
- 5.8 RPC Protocol Interface . . . 92

SYN0014333

**6 Customizing VEM** **97**
- 6.1 Introduction . . . 97
- 6.2 VEM Default Parameters . . . 97
- 6.3 VEM Key, Menu, and Command Bindings . . . 107
- 6.4 Technology Information . . . 107

**7 Creating a New Technology** **109**
- 7.1 Display Attributes . . . 109
- 7.2 Design Rules . . . 111
- 7.3 Technology Properties . . . 112
- 7.4 Palettes . . . 114
- 7.5 Testing a Technology Description . . . 117
- 7.6 Mask manipulation scripts . . . 118
- 7.7 Special tools . . . 118
- 7.8 Creating a Cell Library in Oct . . . 118
- 7.9 Extra Annotation for Tools . . . 120

**8 Berkeley Logic Interchange Format (BLIF)** **125**
- 8.1 Modules . . . 125
- 8.2 Logic Gates . . . 126
- 8.3 Module (subcircuit) references . . . 126
- 8.4 PLA References . . . 126

# Chapter 1

# Getting Started

This chapter contains general information about running and installing the Octtools. It assumes some familiarity with UNIX.

## 1.1 Setting Up the UNIX Environment

To use the Octtools, you will need two things defined in your UNIX environment:

- the OCTTOOLS environment variable; and
- the Octtools binary directory in your search path.

Additionally, some tools require general environment variables that some UNIX systems do not initialize, such as DISPLAY (for X) and USER. On such systems these variables will have to be explicitly defined.

### 1.1.1 The OCTTOOLS variable

This variable should be set to the machine specific directory in which the tools (binary executables and libraries) reside. This directory can be found directly under the Octtools root directory (after the tools have been installed). Machine platforms, which are determined by both processor architecture and version of UNIX, include:

- hpux (Hewlett-Packard HP-UX Version 8 or later and PA-RISC)
- mips (DEC Ultrix and MIPS)
- sun4 (Sun SunOS 4.X and SPARC)

For example, if you are running on a SPARCstation 2, and the Octtools reside in /usr/cad/octtools, you should put

```
if ( ! $?OCTTOOLS ) then
    setenv OCTTOOLS /usr/cad/octtools/sun4
endif
```

in your .cshrc file.

### 1.1.2 The Octtools bin directory

In order for UNIX to find the Octtools programs, you will need the Octtools bin directory in your search path. The bin directory is found in the OCTTOOLS directory, so

```
set path=($path $OCTTOOLS/bin)
```

in your .cshrc file will add the bin directory to your search path.

## 1.2 Documentation

On-line documentation is available with the **octman** command, a slightly modified version of UNIX **man** that retrieves Octtools documentation. Man pages exist in various degrees of detail and accuracy for most tools. There is also information on Oct library routines (**octman** 3) and formats (**octman** 5). In addition, less formally organized documentation exists on a variety of subjects (**octman** doc).

## 1.3 Installation

To install the Octtools you need:

- a standard C compiler (from HP, Sun, or DEC – gcc is problematic);
- X Windows, Version X11R4 or later; and
- the Athena Widget Set (`libaw.a`) for X, which is not distributed on all machines. You may have to build it yourself, in which case you can get the source via FTP from gatekeeper.dec.com.

In addition, if you want to build the **vov** tool, you need g++.

Installation is a four stage process:

- getting and setting up the tar files of the release;
- unpacking the tar files;
- setting up the compilation environment for the install; and
- compiling and installing the tools.

### 1.3.1 Getting the tar files

To get a copy of the Octtools release package contact

```
software@hera.berkeley.edu
```

or

```
Software Distribution Office
Industrial Liaison Program
205 Cory Hall
University of California at Berkeley
Berkeley, CA 94720
(510) 643-6687
(510) 643-6694 (fax)
```

The release package consists of a few text and script files (including `ARCHITECTURE`, `COPYRIGHTS`, `INSTALL`, `PREINSTALL`, and `README`), and a number of compressed tar files that contain source code, technology files, and documentation. All these files should be put in the Octtools root directory. The files `ARCHITECTURE`, `INSTALL`, and `PREINSTALL` are scripts that must be executable (chmod 755 ...).

### 1.3.2 Unpacking the tar files

The `PREINSTALL` script unpacks the tar files. It takes an argument, either `KEEP` or `REMOVE`. If `KEEP` then the compressed tar files are kept; if `REMOVE` then the tar files are removed and just the unpacked files are kept, saving disk space.

The `PREINSTALL` script also verifies checksums on the tar files.

Note that the release package includes items that are not directly part of the Octtools, such as Spice. These items are not unpacked or installed automatically. They also have their own documentation.

### 1.3.3 Setting up the compilation environment

The INSTALL script sets up the platform dependent binary and library directories, and compiles and installs the tools. Two items must be set up correctly for the script to work.

- The OCTTOOLS environment variable must be defined. If it is undefined the INSTALL script will try to set it, but it is advisable for the variable to be defined in advance. See above for how to set it.

- Two configuration variables in INSTALL determine the location of the X11 include files and libraries, xincloc and xlibloc. As delivered, these are set to values reasonable at UCB, but will almost certainly have to be changed for your system.

In addition, the ARCHITECTURE script must run correctly. Run the script and see if it returns the type of machine you are using. If it does not you can either fix the ARCHITECTURE script or fix INSTALL to compile for your machine type without using ARCHITECTURE.

### 1.3.4 Building the tools

The INSTALL script uses a layered sequence of makefiles, with one in the OCTTOOLS directory, one in the src directory, one in each tool type directory (such as Layout or Framework), and finally one for each tool. Each of the general (non tool specific) makefiles has an associated TOOL-LIST file that is used to select which tools are built. These files can be edited to change the selected tools. The default TOOL-LIST files make everything but BIGOCT, BearFP, LightLisp, OctLisp, Unsupported, Vov, boss, and slide-maker.

Once the environment is configured, simply run INSTALL to compile and install the programs, program libraries, technology descriptions, and documentation.

All object files and other effluvia can be removed by connecting to the OCTTOOLS directory and typing make clean.

## 1.4 How the Octtools Are Organized

The initial release, before installation, resides in the Octtools root directory under common, in the four directories

- tech – technology files;
- lib – assorted design-related library items;
- document – ~~the PostScript form~~ of the manual~~,~~ and
- src – program sources, man pages, test cases, etc.

The installation process creates

- common/include (from include files scattered through common/src)
- common/doc (from doc files scattered through common/src)
- common/man (from man files scattered through common/src)
- OCTTOOLS/<architecture> (the platform-specific directory)
- OCTTOOLS/<architecture>/lib (which holds lib*.a files)
- OCTTOOLS/<architecture>/bin (which holds executables)
- links from OCTTOOLS/<architecture> to various directories in common

The Octtools universe residing in `common/src` is conceptually divided into two basic classes of things: libraries which, when compiled, end up in `<architecture>/lib`, and tools, which use the libraries and, when compiled, end up in `<architecture>/bin`. Almost all the libraries (called packages), are found in the directories

```
common/src/Packages
common/src/Xpackages .
```

and the tools are in

```
common/src/Framework
common/src/Synthesis
common/src/Layout
common/src/Technology
common/src/Simulation
common/src/OtherTools
common/src/Utils
```

**Packages:** This directory, along with Xpackages and Technology, contains the source code for the libraries. Important packages here are **oct** (the basic Oct database routines), **oh** (additional Oct helper functions and macros), **mm** (the Oct versions of malloc and free, which don't port very well), and **port** (which contains items related exclusively to porting, primarily `port.h`; note that not all porting problems are solved in `port.h` – individual tools also conditionally compile specialized code). There are several other data structure packages as well, such as **st** (symbol table) and **avl** (avl tree). Most of these packages are documented; either look in `common/doc` or use **octman** (which looks in the `common/doc` directory).

**Xpackages:** This directory contains X libraries primarily used by **vem**, including **dds** (the dialog box definition system) and the rpc libraries used by vem clients.

**Framework:** This directory contains general tools and utilities, some of which are oriented toward design, and some of which are Oct database maintenance utilities. Tools include **vem** (the graphical editor), **oct2ps** (a postscript translator), **octman** (man for the Octtools), **bdnet** (the netlist front end), **fabprep** (which prepares an Oct layout for fab), **chipstats** (which compiles statistics on a chip), **octflatten** (which flattens an Oct cell hierarchy), and **attache** (the general Oct facet database editor). These tools range from the almost trivial (**inconsistent**) to the gigantic (**vem**).

**Synthesis:** This directory contains logic and PLA synthesis tools, including **espresso** and **misII**.

**Layout:** This directory contains the layout tools, most notably **wolfe/Timberwolf** (the standard cell placer and router) and **mosaico** (the macro cell router), which is a shell script that in turn invokes several other tools.

**Technology:** The primary item in this directory is **tap**, the technology access package, which is used (by **vem**) to access technology files.

**Simulation:** The primary items in this directory are **musa**, the multi-level simulator that runs off of the Oct data base, and **steps**, the Oct to Spice translator.

**OtherTools:** This directory contains miscellaneous items such as **xgraph** and **slide-maker**.

**Utils:** This directory contains assorted items used for building an Octtools distribution.

## 1.5 Making a Tool

If you need to build an individual tool, cd to the tool's directory, fix `XINCLOC` and `XLIBLOC` in the Makefile to be your X directories (see above), set any necessary platform flags in the CC line, for example

```
CC =/bin/cc -Dhpux
```

and type

```
make <tool-name>
```

or

```
make install
```

to put the executable in `OCTTOOLS/bin`, or

```
make debug-g
```

to get a version with symbols usable by the debugger. If you want a debug-g version, you will also need debug-g versions of the packages (see below).

To build a package (library), cd to the package's directory, fix the X directories, and type

```
make install
```

(which will put the package in `OCTTOOLS/lib`), or

```
make debug-g
```

to get a debuggable version (which also puts the package in `OCTTOOLS/lib`).

In general, in a tool or package directory you can type

```
make help
```

for a list of the actions **make** knows about. For more details on Octtools makefiles, if you need to modify or create one, try

```
octman create-octtools-makefile
```

# Chapter 2

# The General Structure of Oct

OCT is a data manager for VLSI/CAD applications. It is a major component of the Berkeley CAD framework that has been used at Berkeley (and other sites) for the last few years[3]. It offers a simple interface for storing information about the various aspects of an evolving chip design. A basic unit in a design is the *cell*. A cell is a portion of a chip that the designer wishes to consider as a unit. This can be as small as a transistor or a NAND gate, or as large as the entire floorplan of a CPU. A cell can consist of instances of others cells, such as a NAND gate consisting of several transistors or the floorplan consisting of an ALU, a register file, *etc*.

A cell can have many aspects or *views*, depending on what point in the design you are at and on what design style you use. There can be a schematic view, showing in an abstract way what sub-cells the cell consists of and how they are connected. There can be the symbolic view, where additional information of rough relative placement, sub-cell size and shape, and initial implementation of interconnect might be kept. And even more refined, the physical view, where the implementation is fully defined with exact placement and specific geometry. In addition there are quite different views, such as the simulation view, which might contain the description of the cell in a format that a particular simulator could understand. What views a cell may have, what is contained in those views, and how they are related are not issues addressed by OCT; these decisions are left to authors of the tools that use OCT.

Finally, there is the concept of *facets*. Cells are hierarchical, *i.e.* they contain instances of other cells which in turn may contain instances of other cells, *etc*. For various applications, you may wish to cut off this hierarchy; instead of continuing to traverse the hierarchy by processing the contents of a view, you represent the cell by some simplified abstraction. For a graphics editor the abstraction might be a bounding box, for a routing tool the abstraction might consist of the terminals and routing regions of the cell, and for a design rule-checker it might be just that geometry on the boundary of the cell that needs to be checked against neighboring cells (thus avoiding rechecking the interior of the cell each time it is instantiated). We call these various abstractions *facets* of the view.

Each view has a facet named *"contents"* which contains the actual definition of the view, and then various application-dependent interface facets. Again, OCT does not define what interface facets may exist, and what their relation to the contents facet might be. The only facet OCT has any explicit knowledge about is the *contents* facet, which is the facet that defines the name and number of external (formal) terminals a view has. When an instance of a view or a new interface facet is created, the *contents* facet of that view is consulted to find out what terminals the instance or facet inherits from the view. The *contents* facet must be created before any other facet is created and before any instance of that view is created.

The final specification for a facet is the version. The version allows there to be multiple time sequenced facets. Versions are generally ignored by most OCT applications and are only dealt with by data management systems.

The facet is the fundamental unit that you edit in OCT. A particular facet of a view of a cell is opened and edited independently of the other facets of that view and the other views of that cell. The facet consists of a collection of objects that are related by *attaching* one to another (see Figure 2.1). A box is put on a particular layer by attaching it to that layer, a terminal is shown to be part of a net by attaching it to that net, a box is shown to implement a terminal by attaching it to that terminal. Objects can be attached to more than one object and can have more than one object attached to them.

Being attached is not a particularly strong property: if you delete an object, you do not delete objects attached to it, you merely detach the connection. The structure created by attachments can be thought of as a directed graph of OCT objects. There are a few limitations on what can be attached to what (see the description of octAttach for details).

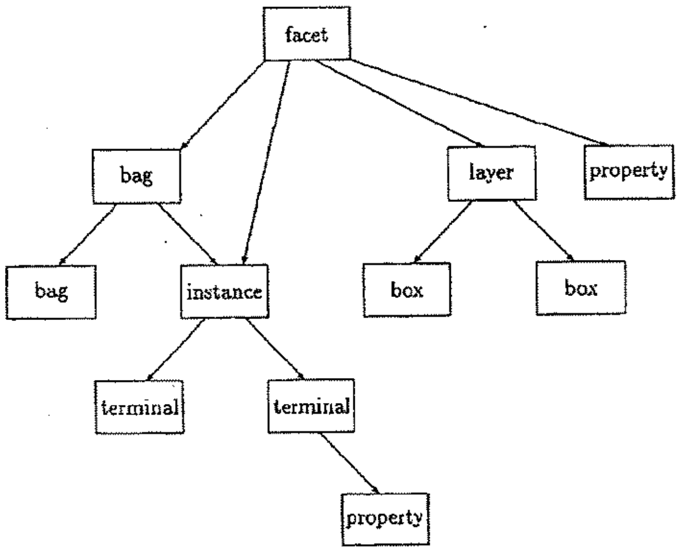

Figure 2.1: Sample Attachments in OCT.

OCT provides a *mechanism* for representing data but places no meaning on the data. *Policy* is used for assigning meaning to the data represented using OCT. For example, OCT has objects that represent layers and geometry, but does not specify how the objects are related to give the *meaning* of a geometry implemented on a given layer. The *policy* states that a geometry that is contained by a layer is implemented on that layer. As another example, OCT has objects that represent nets and terminals, but does not specify how connectivity is represented. The *policy* describes how terminals and nets are used to represented connectivity. See chapter 3 for detailed information about specific OCT policies.

## 2.1 The OCT Objects

The objects that can be contained in a facet are described by the C language structure octObject:

```
struct octObject {
    octObjectType type;
    octId objectId;
    union {
        struct octFacet facet;
        struct octInstance instance;
        struct octProp prop;
        struct octTerm term;
        struct octNet net;
        struct octBox box;
        struct octPolygon polygon;
        struct octCircle circle;
        struct octPath path;
        struct octLabel label;
        struct octBag bag;
        struct octLayer layer;
        struct octPoint point;
        struct octEdge Edge;
    } contents;
};
```

`octObject` is used to describe all of the things contained in the database. The `type` field describes what type of object is contained, and the `contents` field is the union which actually contains the data. Thus if you have an object `obj` that describes a box, `obj.type` has the value `OCT_BOX` and the description of the box is in `obj.contents.box`.

### objectIds

Along with the data describing the object and the type of the object, each object contains a field `objectId`. The database uses this ID to find the internal database description of the object. In addition, given the object ID, `octGetById` will return the associated object. This means the object ID can be used as form of pointer to the object, so the entire object need not be kept to reference it, only the id is necessary. These ids are guaranteed to be unique during an OCT session. When an object is deleted, its ID is marked as invalid and may not be used in any further operations; ID's are not reused.

ID's happen to be 32-bit integers, but it is poor programming practice to rely on this fact. ID's should be handled only with the OCT functions `octIdsEqual()`, `octIdIsNull()`, `octHashId()`. It is quite likely that in future versions of OCT ID's will be structures with 64 bits or more.

### externalId

As opposed to the `objectId`, which is unique across all objects in all facets currently in memory and persists only as long the the object is in memory, the externalId (xid) persists across sessions. Xids are unique within a facet and combined with the facet that contains the object, the xid is unique across all objects in all facets. The xid is not part of the user-visible OCT object structure, but can be accessed via the ~~function~~ `oct~~External~~Id`.

### Units

Coordinates in the database, `octCoord`, have no units associated with them. They are simply integers of a reasonable size. It is up to policy to define the units.

## OCT_FACET

```
struct octFacet {
    char *cell;
    char *view;
    char *facet;
    char *version;
    char *mode;
};
```

`octFacet` describes the `facet` facet of the `view` view of the cell `cell`. The facet is to be opened for read, write (and delete old contents, if any), or append, depending on whether `mode` is "r", "w", or "a" respectively. The `version` field is ignored for the moment and should be set to OCT_CURRENT_FACET.

## OCT_POINT

```
struct octPoint {
    octCoord x;
    octCoord y;
};
```

`octPoint` describes a point at the coordinates x, y.

## OCT_BOX

```
struct octBox {
    struct octPoint lowerLeft;
    struct octPoint upperRight;
};
```

`octBox` describes a manhattan (all sides parallel to the x or y axes) rectangle. It has its lower left corner (smallest x and y) at the point `lowerLeft` and upper right corner (largest x and y) at the point `upperRight`.

## OCT_CIRCLE

```
struct octCircle {
    octCoord startingAngle;
    octCoord endingAngle;
    octCoord innerRadius;
    octCoord outerRadius;
    struct octPoint center;
};
```

`octCircle` actually describes an arc starting at `startingAngle` (which is in tenths of degrees measured counter-clockwise from the x-axis) and continuing counter-clockwise to `endingAngle`. The inside edge of the arc is at a distance `innerRadius` from `center`, while the outside edge is `outerRadius` away. `OuterRadius` should be greater than `innerRadius`. A simple full circle would have `endingAngle` equal to ``startingAngle + 3600'', `outerRadius` equal to the radius of the circle and `innerRadius` equ~~al to~~ zero.

## OCT_PATH

```
struct octPath {
    octCoord width;
};
```

`octPath` describes a 'wire' of width `width`. The path followed by the wire is described by an array of points that is set and accessed indirectly, using the functions `octGetPoints` and `octSetPoints`. Consult the man pages for those functions for further details.

## OCT_POLYGON

```
struct octPolygon {
    int  unused;        /* Dummy field, just to avoid having an empty struct. */
};
```

octPolygon describes a polygon. The vertices themselves are accessed the same way as the points in octPath are. The first and last vertices are assumed to be connected. The polygon can be self-intersecting, with the interior being those points about which the polygon has non-zero winding number. Note that since a polygon is described entirely by its vertices, its definition contains only a dummy field that should never be used. It has been included in OCT to maintain symmetry with the other objects, and because we had doubts about the portability of an empty structure definition.

## OCT_EDGE

```
struct octEdge {
    struct octPoint start;
    struct octPoint end;
};
```

octEdge describes an edge (a zero width line) extending from the point start to the point end. It is included for the use of programs like compactors that may wish to represent constraints between edges of various geometries.

## OCT_LABEL

```
#define OCT_JUST_BOTTOM 0
#define OCT_JUST_CENTER 1
#define OCT_JUST_TOP 2

#define OCT_JUST_LEFT 0
#define OCT_JUST_RIGHT 2

struct octLabel {
    char *label;
    struct octBox region;
    octCoord textHeight;
    unsigned vertJust : 2;
    unsigned horizJust : 2;
    unsigned lineJust: 2;
};
```

The octLabel object is used to represent arbitrary text annotation. This text, stored in the field label, may be any number of lines each separated by a newline character ("\n"). The position of the resulting block of text is given by the box region. The height of each line of text is given by the OCT coordinate textHeight. The block of text as a whole may be justified within the box region. The bit-field vertJust specifies the vertical justification of the text block (one of OCT_JUST_BOTTOM, OCT_JUST_CENTER, or OCT_JUST_TOP). The bit-field horizJust specifies the horizontal justification of the text block (one of OCT_JUST_LEFT, OCT_JUST_CENTER, or OCT_JUST_RIGHT). Finally, each line of text may be horizontally justified relative to the text block. The bit-field lineJust specifies this justification (one of OCT_JUST_LEFT, OCT_JUST_CENTER, or OCT_JUST_RIGHT). Graphically, the layout of the octLabel object is shown in Figure 2.2.

## OCT_NET

```
struct octNet {
    char *name;
    int32 width;
};
```

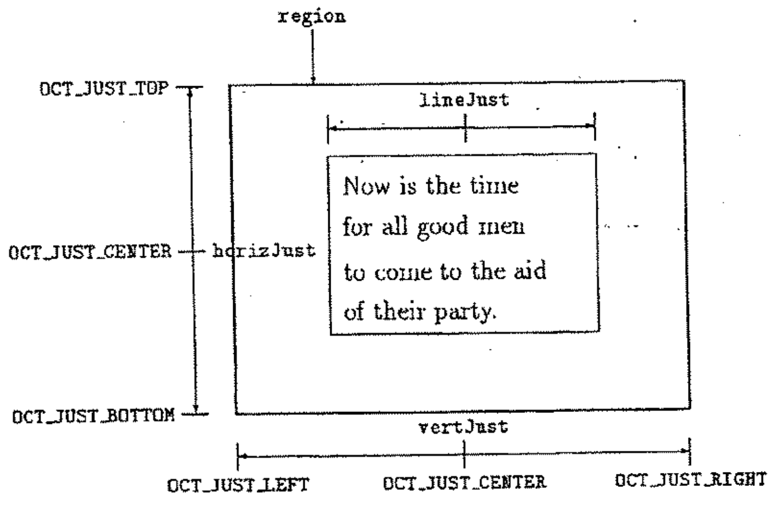

Figure 2.2: The octLabel Object

octNet describes a net with optional name name. The width can be used to represent a vector of nets. A net represents a logical connection between the terminals that it contains. In addition it can contain geometry and instances so that the physical implementation can be associated with it. Terminals, geometry, instances and nets are attached and detached using octAttach and octDetach. They can be accessed using octInitGenContents and octGenerate.

## OCT_TERM

```
struct octTerm {
    char *name;
    octId instanceId;
    int32 formalExternalId;
    int32 width;
};
```

octTerm describes a terminal of an instance or of a view. If it is a terminal of an instance (an actual terminal), then instanceId is the octId of the instance to which it belongs. If it is a terminal of the current facet (a formal terminal), then instanceId is equal to oct_null_id. You cannot explicitly create an actual terminal. They are created implicitly created when an instance is created, with an actual terminal being created in the instance of a view for each formal terminal in the view itself. Formal terminals can only be created in the contents facet of a view, and all other facets may only reference them. The width can be used for representing a vector of terminals. To find which nets a terminal is connected to, use octInitGenContainers and octGenerate. In an actual terminal, the formalExternalId is the xid of the formal terminal that the actual terminal represents. The xid of the formal terminal is the same in all facets in the view.

## OCT_BAG

```
struct octBag {
    char *name;
};
```

octBag describes an object whose only purpose is to hold other objects. As usual, objects are attached and detached from the bag using octAttach and octDetach, while its contents are retrieved using octInitGenContents and octGenerate. A bag might be used to hold the currently selected items for a graphics editor, the objects on the critical path in a timing simulator, or any other group of objects that should be associated.

## OCT_LAYER

```
struct octLayer {
    char *name;
};
```

octLayer describes the layer object for layer name. Geometries must be explicitly attached to one (or more if really necessary) layer using octAttach.

In order to retrieve (that is, in OCT lingo, to "generate") all the geometry on one layer a programmer can take a layer object (obtained either by generating all layer objects in the facet or by using octGetByName to get a particular layer) and use octInitGenContents and octGenerate to generate all or part of the geometry attached to that layer. Note that since layers are treated like any other object, it is possible to have more than one layer in a facet with the same name. If you wish to avoid creating a layer with the same name as an already existing layer, you can use the function octGetOrCreate to retrieve a layer with that name if it already exist, or to create it and attach it to the facet if it does not yet exist.

## OCT_PROP

```
enum octPropType {
    OCT_NULL, OCT_INTEGER, OCT_REAL, OCT_STRING, OCT_ID, OCT_STRANGER
    OCT_REAL_ARRAY, OCT_INTEGER_ARRAY
};

struct octProp {
    char *name;
    octPropType type;
    union {
        long integer;
        double real;
        char *string;
        octId id;
        struct octUserValue stranger;
    } value;
};


struct octUserValue {
    int length;
    char *contents;
};
```

octProp describes the properties that can be attached to any object (including another property). The property is given the name name and, depending on the value of type, value can be an integer, a floating point number, a string, an octId, an array of integers, an array of real, or any collection of bytes as defined in octUserValue. Properties can be used to attach any type of information to an object, anything from the last time it was modified to the text of the code that generated it. Because properties can have properties, arbitrarily complex structures can be built.

The value of a property on a object can be obtained by using octGetByName, a new property can be added by using octAttach, and it can be changed by using octModify. Since properties are no different than any other object, it is possible (and sometimes useful) to attach two properties to an object with the same name. To avoid this (if you so desire), you can use the function ''octCreateOrModify(container, obj)'', which only creates and attaches obj to container if it doesn't already exist, otherwise the existing object is modified.

## OCT_INSTANCE

```
struct octInstance {
    struct octTransform transform;
    char *name;
    char *master;
    char *view;
    char *facet;
    char *version;
};

enum octTransformType {
    OCT_NO_TRANSFORM = 0,        /*  x ->  x, y ->  y */
    OCT_MIRROR_X = 1,            /*  x -> -x, y ->  y */
    OCT_MIRROR_Y = 2,            /*  x ->  x, y -> -y */
    OCT_ROT90 = 3,               /*  x -> -y, y ->  x */
    OCT_ROT180 = 4,              /*  x -> -x, y -> -y */
    OCT_ROT270 = 5,              /*  x ->  y, y -> -x */
    OCT_MX_ROT90 = 6,            /*  x -> -y, y -> -x */
    OCT_MY_ROT90 = 7,            /*  x ->  y, y ->  x */
    OCT_FULL_TRANSFORM = 8
};

struct octTransform {
    struct octPoint translation;
    octTransformType transformType;
    double generalTransform[2][2];
};
```

octInstance describes an instance named name of the facet facet of the view view of cell master transformed by the transform transform. The facet field is important because it determines the size of the instance bounding box. In most layout the bounding box of the interface facet is identical to the bounding box of the contents facet, but this is not necessarily true for example in schematics, where a large circuit can consists of many transistors and so have a large bounding box in the contents facet, while it may be represented by a small rectangle in the abtraction contained by the interface facet.

The transform consists of a translation and either 1 of the 8 eight manhattan transforms defined in octTransformType or a general 2 by 2 tranformation. Instantiating an instance also causes the terminals for that instance to be created as well. See octTerm above.

Inside OCT there are routines that check the consistency of the terminals in an instance and the terminals in the contents facet of its master. These routines are activated whenever you open a facet, for all the instances contained in that facet (of course, masters with multiple instances are checked only once). If a facet has many instances of many different types, this checking process may take a long time (a few seconds).

Some tools need to open the master whenever they find an instance. For example, vem needs to do that in order to find out how to display the instance. In general, the interface facet is much smaller than the contents facet, and therefore it is much faster to open. If you want to increase the efficiency of your tools, you need to have some understanding of the mechanism used by OCT to maintain terminal consistency (another example to study is offered by the tool tapPP: `octman tapPP`).

## OCT_CHANGE_LIST

```
typedef int32 octObjectMask;
typedef int32 octFunctionMask;

struct octChangeList {
    octObjectMask objectMask;
    octFunctionMask functionMask;
};

OCT_CREATE_MASK
OCT_DELETE_MASK
OCT_MODIFY_MASK
OCT_PUT_POINTS_MASK
OCT_DETACH_CONTENT_MASK
OCT_DETACH_CONTAINER_MASK
OCT_ATTACH_CONTENT_MASK
OCT_ATTACH_CONTAINER_MASK

OCT_ATTACH_MASK         ('OR' of the ATTACH masks)
OCT_DETACH_MASK         ('OR' of the DETACH masks)

OCT_ALL_FUNCTION_MASK   ('OR' of all masks)
```

`octChangeList` objects are used for monitoring changes in an OCT facet. For each set of operations and objects that is to be monitored, a change list is created in the facet with the `objectMask` field set to the 'OR' of the masks of the object types to be monitored and the `functionMask` field set to the 'OR' of the masks of the operation types to be monitored. For example, the following would monitor all creation and deletion operations on OCT_BOX and OCT_PROP objects:

```
octObject facet, cl;

cl.type = OCT_CHANGE_LIST;
cl.contents.changeList.objectMask = OCT_BOX_MASK
                                  | OCT_PROP_MASK;
cl.contents.changeList.functionMask = OCT_CREATE
                                    | OCT_DELETE;
octCreate(&facet, &changes);
```

See Section 2.10 for more information.

## OCT_CHANGE_RECORD

```
struct octChangeRecord {
    int32 changeType;
    int32 objectExternalId;
    int32 contentsExternalId;
};
```

`octChangeRecord` objects are created and attached to `octChangeList` objects whenever a monitored change takes place. See Section 2.10 for more information.

B

# Octtools 5.2
# Part I: User Guide

Electronics Research Laboratory
University of California at Berkeley

May 25, 1993

```
musa0> mv ps ps<3:0>
musa0> mv ns ns<3:0>
musa0> ma in
> se ps $1
> ev
> sh "%h %h" ps ns
> $end
musa0> in h0
 ps:H0   ns:H1
musa0> in h1
 ps:H1   ns:H2
musa0> in h2
 ps:H2   ns:H3
musa0> in h3
 ps:H3   ns:H4
musa0> in h4
 ps:H4   ns:H5
musa0> in h5
 ps:H5   ns:H6
musa0> in h6
 ps:H6   ns:H7
musa0> in h7
 ps:H7   ns:H8
musa0> in h8
 ps:H8   ns:H9
musa0> in h9
 ps:H9   ns:H0
musa0> in ha
 ps:HA   ns:H0
musa0> quit
```

The counter control logic is now complete.

## Flip-Flops

We will use flip-flops from the MSU (Mississippi State University) library for this design. We can create an OCT cell with 4 flip-flops using the netlist to OCT translator, bdnet. To do this, create a file, `reg.bdnet` with the following contents:

```
model reg:logic;

TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT reset;
INPUT d<3:0>;
CLOCK clk;
SUPPLY Vdd;
GROUND GND;
OUTPUT q<3:0>;

ARRAY %x from 0 to 3 of
INSTANCE $OCTTOOLS/tech/scmos/msu/stdcell2_2/dfrf301:physical
     DATA1:d<%x>;
     CLK2:clk;
     RST3:reset;
     Q:q<%x>;
     "Vdd!":Vdd;
     "GND!":GND;
ENDMODEL;
```

The OCT view `reg:logic` is created with the command:

```
% bdnet reg.bdnet
```

Again, it would be a good idea to simulate the cell using **musa**. A sample simulation run is shown below. (Musa commands entered by the user in the sample run are preceeded by the

C

# BDSYN Users' Manual

Version 1.1

Russell B. Segal
University of California
Berkeley

May 22, 1987

## 1 Introduction

Bdsyn is a hardware description translator. It takes as input a textual description of a block of combinational logic and produces a collection of logic functions which implement the described function. Bdsyn was written to be a front end to the Berkeley Synthesis System. Work on bdsyn was begun in the spring of 1986 as a class project for Berkeley's synthesis class. For the synthesis class, we were given permission to use Digital Equipment Corporation's proprietary multi-level simulator decsim. The functional simulation language bds is used for high level simulation in decsim. We have chosen to use a subset of the bds language as the input language to bdsyn. The output of bdsyn is a multiple-level logic representation of the described function. The output created is in blif (Berkeley Logic Interchange Format). This output is suitable input for mis which is the multiple level logic minimizer in the Berkeley Synthesis System.

### 1.1 What Bdsyn Is

Bdsyn is a tool for quickly describing and implementing combinational logic. It allows the user to describe a circuit function and produce an implementation of that circuit automatically. Bdsyn's output is given in a "multiple-level" format. This means that bdsyn is capable of translating many types of logic configurations that would be impractical to represent in a two-level pla form. Bdsyn supports many high-level language constructs: subroutines, if--then--else, select, constant bounded for loops, multiple assignment to variable, and multiple returns from a single routine. In addition, bdsyn supports complex operators such as addition, and shifts by a variable amount. (Note that complex operators may imply a large number of gates.)

When bdsyn is used in conjunction with mis, it can be used to describe logic in standard cell, pla, or a more complicated module generated form.

## 1.2 What Bdsyn Is Not

It is very important to keep in mind that bdsyn can be used for describing combinational logic blocks only. We define combinational logic to be any block of logic which does not use system clocks or signal latching of any kind. Combinational logic does not hold state information. In particular, describing a finite state machine in bdsyn requires the logic designer (or a program) to add external latches to the described combinational logic block that hold the current state of the machine.

The limitation of describing only combinational logic is common to pla description programs as well as bdsyn. (Although there may be provisions for automatically adding the required external latches.) The reason we find it necessary to describe this limitation in such detail here, however, is that some of the bdsyn constructs (e.g. for loops and multiple assignment) appear very sequential in nature. In reality, they are merely interpreted as a short hand for describing parallel structure. There are many examples of the use of sequential-like descriptions in the last section of this guide.

# 2 Using Bdsyn

## 2.1 Describing Logic in Bdsyn

Bdsyn descriptions are very similar to standard procedural programming languages (Pascal, C, etc.) Describing logic in bdsyn amounts to writing a program that accomplishes the desired function. Each flow-of-control structure and operation corresponds roughly to a small block of logic. Each variable corresponds to "bit vector," which is a group of single bit signals in the hardware.

At the top of the bdsyn description it is necessary to define input and output "ports." In a programming point of view, these ports can be viewed as external variable declarations. In a hardware point of view, these ports represent primary input and outputs of the combinational logic block. Once the ports are declared, the logic designer should define a set of values for the input ports for which he wishes the hardware to operate. If he then writes a program that will generate desired output values for each of the interesting input combinations, this constitutes a legal description.

To say it in a different way, suppose a bdsyn "program" has been written. Bdsyn will create logic that will act in an identical manner to that program. If a set of inputs were to be introduced to the bdsyn program and the program were executed (run in a software sense), a particular set of outputs would be generated. If these same inputs were introduced to the combinational logic block created by bdsyn, the same set of outputs would result.

## 2.2 Running Bdsyn

Once a hardware description has been written, it can be converted to logic by running bdsyn. The input description is read from the file given on the command line. The blif output is returned to standard output. There are several command line options for bdsyn which are outlined in the bdsyn manual page. Two of the options, however, will be discussed here.

The -b option tells bdsyn not to do its cleanup evaluation. This cleanup evaluation process in bdsyn eliminates redundant and unnecessary logic. Unfortunately, it may also have the undesired effect of eliminating user specified intermediate variables that bdsyn considers unimportant. The -b makes sure that the variable are preserved, at the price of extra logic.1

The -c option is followed by a number which tells bdsyn how much "collapsing" of logic that it should do. Collapsing is done in order to decrease the amount of output which bdsyn creates. Although quite rare, it is possible to create an input description which causes bdsyn to collapse too much logic and run for inordinate amounts of time. The option -c1 will limit the amount of collapsing that is done, and will alleviate the problem. Bdsyn uses mis do to logic collapsing for it. The actual communication with mis is accomplished through the use of unix pipes. On non-unix systems or on a system where mis is not available, the option -c0 should be used to turn off collapsing altogether.

## 2.3 What is New in Version 1.1

There are two main changes in bdsyn since version 1.0 that users may notice. The first change is that the intermediate variable names created by bdsyn are shorter. Each variable is followed by two numbers. The first

-------------------------

1The extra logic could easily be reduced by MIS at a later time.

number is the block in the description in which the variable is set. The second number represents how many times the variable has been revised in the block.
The second change is that bdsyn will create its own intermediate variables corresponding to routine return values and condition expressions from if and select statements. These variables are good candidates for having a large fanout. They are created in the hope that it will save mis some work in extracting common factors. On occasion, bdsyn may create an internal variable which does not fan out. mis will give a warning when such a variable is created. In general, it is safe to ignore these warnings for variables whose name begin with "$$COND."

## 3 Language Constructs

### 3.1 Names and Numbers

Bdsyn accepts a subset of the bds language as input. bds is a Pascal-like language. The input consists of names, numbers, and reserved keywords separated by white space. White space is spaces, tabs, and carriage returns.

Any characters following a `!' (exclamation point) are interpreted as comments by bdsyn. Legal names may contain the characters a-z, A-Z, 0-9, `' (underscore), and `%'. Other characters are legal in the name as long as the name is quoted with double quotes ("). Names may not be any of the reserved keywords (given below) unless they are quoted. Names that begin with `$' are illegal. Bdsyn, like decsim, is case insensitive, so the name "signal" is the same as the name "SIGNAL."
Numbers in bdsyn must be positive and may be given in any one of base 2, 4, 8, 10, or 16. Decimal numbers are represented as a simple integer (e.g. 511). Other base numbers are represented as an integer followed by a `#' and the base (e.g. 1FF#16). The width, in bits, of a decimal number is the minimum number of bits required to represent that number. The width of a number with a `#' is equal to the number of bits required to represent a number in that base with the given number of digits. For example 001#2 has width three, 123#8 has width nine, and 12345678#16 has width thirty-two. Bdsyn has the limitation that numbers in the input must be thirty-two bits or fewer. This limitation may be overcome by the use of the concatenation operator. (e.g. 12#16 & 34567890#16

is a forty bit constant.)

| | | | | |
|---|---|---|---|---|
| and | begin | buf | by | constant |
| do | downto | else | end | endmodel |
| endroutine | endselect | endselectall | endselectone | eql |
| eqv | for | from | geq | gtr |
| if | leave | leq | lss | macro |
| mod | model | nand | neq | nor |
| not | or | otherwise | oxt | require |
| return | routine | select | selectall | selectone |
| sl0 | sl1 | slr | sr0 | sr1 |
| srr | state | sxt | synonym | then |
| to | width | xor | zxt | |

Table 1: Bdsyn Reserved Words

| | | | | |
|---|---|---|---|---|
| behavior | endbehavior | endmodule | entry | forward |
| fourstate | halt | inform | input | invalid |
| module | noentry | output | port | repeat |
| revision | static | twostate | until | while |

Table 2: Keywords Ignored by Bdsyn

## 3.2 Reserved Keywords

Table 1 gives a list of reserved words for bdsyn. These may not be used as names unless they are quoted. In addition to the reserved words there are several keywords which bdsyn explicitly ignores but are included so that complete bds descriptions (not just our subset) may be parsed. They are listed in table 2

## 3.3 Variables

Bdsyn has two types of variables which it understands. The first, "logic variables," imply actual logic and can be thought of as wires in the actual layout. Logic variables are declared with bit subscripts that define their width. The lower bit numbers correspond to the less significant bits, and bdsyn requires that the high bit be first.

```
    STATE name1<7:0>;
```

If the bit range of a variable is omitted in its declaration,

```
    STATE name2;
```

bdsyn will assume a single bit vector (name2<0>).

When using a logic variable (as opposed to specifying one), the bit subscript is used to choose certain bits from the bit vector. If the bit vector is left out, the the entire bit vector is used.

In addition to logic variables, bdsyn defines "meta-variables." Their main application is as the index variable for for loops. Loop index variables are required to be meta-variables. Meta-variables are declared with null bit subscripts:

```
    STATE name3<>;
```

Meta-variables do not imply any logic, and are used for temporary storage of constants. When meta-variables appear on the left hand side of an assignment, the right hand side is evaluated and assigned to the meta-variable. The right hand side of this assignment must be a meta-variable expression (i.e., an expression written from meta-variables and constants). When meta-variables appear on the right hand of an assignment, they are treated exactly like constants. This means that meta-variables may be used in places where logic variables are disallowed by bdsyn. Section 6 covers more about meta-variables.

## 3.4 Input Format

The following is a bnf description of a typical input program for bdsyn. Keywords are given in capital letters, and user names and productions are given in small letters. Optional phrases are given in square brackets `[ ]'. Phrases in curly braces `{ }' many be repeated zero or more times in the input. The character `|' (vertical bar) represents a choice of many lines.

The basic bdsyn input file consists of a "model":

```
    MODEL model_name [output_var {, output_var}] =
                                  [input_var {,  input_var}]
;
        {global_declaration ;}
        {routine}
    ENDMODEL [model_name];
```

A "globaldeclaration" is defined as the following:

```
    |   STATE global_var {, global_var}
    |   CONSTANT constant_name = number {, constant_name  =
number}
        |     SYNONYM  var_name  = var_name  {,  var_name  =
var_name}
```

A "routine" is defined as the following:

```
     ROUTINE routine_name [bit_subscripts] [( var_param  {,
var_param} )] ;
          { local_declaration ;}
          { [ label_name: ] statement ;}
     ENDROUTINE [routine_name];
```

A "localdeclaration" is defined as the following:

```
     |   STATE local_var {, local_var}
     |   CONSTANT constant_name = number {, constant_name  =
number}
      |     SYNONYM  var_name  = var_name  {,  var_name  =
var_name}
```

A statement of a routine is defined below. Note: The square brackets in the select statements represent actual brackets in the input.

```
     |   BEGIN {statement ;} END
     |   variable = expression
     |   IF expression THEN statement [ELSE statement]

     |   FOR meta_var FROM const_expr TO const_expr
                             [BY const_expr] DO statement

     |   FOR meta_var FROM const_expr DOWNTO const_expr
                             [BY const_expr] DO statement

       |      SELECT  expression  FROM  {[expression   {,
expression}]: statement;}
                                {[OTHERWISE]: statement  ;}
ENDSELECT

       |      SELECTALL  expression  FROM  {[expression  {,
expression}]: statement;}
                                {[OTHERWISE]: statement  ;}
ENDSELECTALL

       |      SELECTONE  expression  FROM  {[expression  {,
expression}]: statement;}
                                {[OTHERWISE]: statement  ;}
ENDSELECTONE

     |   RETURN [expression]
     |   LEAVE label_name
```

Note that these tables imply that a semicolon `;' separates every statement. As in Pascal, there is no semicolon before else, but unlike Pascal there is a semicolon before end. Expressions and const-expr's (constant expressions) will be described in later sections.

## 3.5 Declaration Statements

The model statement defines the primary inputs and outputs to the combinational logic block being described. The declared "ports" may not be meta-variables.

The state, constant, and synonym may be used as either global declarations, applying to all routines, or local declarations, applying to only one routine. The state statement is for declaration of variables. All variables must be declared prior to use. Local variables can be used only within the scope of a routine and their values are lost between consecutive calls to a routine. Constant declarations are used to assign a name to a constant. Declared constants can be used anywhere a constant can. Synonyms are used to create an alternate name for a previously defined variable. They are particularly useful for defining subfields of a variable:

```
    SYNONYM opcode<6:0> = instruction<31:25>;
```

## 3.6 Flow-of-control Statements

Bdsyn offers a rich selection of flow-of-control statements. They are if, select, selectone, selectall, for, return, and leave. The if statement implements optional execution of a statement. If the least significant bit of the if expression evaluates to one, the then clause is used. If the low bit evaluates to zero, the else clause (if present) is used. The select statement is used to execute one statement out of many. If the expression following the select keyword is equal to the expression appearing in the square brackets, the corresponding statement is executed.

selectone and selectall are simply a shorthand for several if statements. selectone implies nested if statements:

```
    SELECTONE expr FROM
        [expr1, expr2]: statement1;
        [expr3]: statement2;
        [OTHERWISE]: statement3;
    ENDSELECTONE;
```

is equivalent to:

```
    IF (expr EQL expr1) OR (expr EQL expr2) THEN
        statement1
    ELSE IF expr EQL expr3 THEN
        statement2
    ELSE
        statement3;
```

selectall is equivalent to sequential if statements:

```
    SELECTALL expr FROM
        [expr1, expr2]: statement1;
        [expr3]: statement2;
        [OTHERWISE]: statement3;
    ENDSELECTALL;
```

is equivalent to:

```
    IF (expr EQL expr1) OR (expr EQL expr2) THEN
        statement1;
    IF expr EQL expr3 THEN
        statement2;
    IF (expr NEQ expr1) AND (expr NEQ expr2) AND (expr  NEQ
expr3) THEN
        statement3;
```

selectall is different than select in that selectall implies sequentiality in its case evaluation. The select statement can be viewed as having all of its cases evaluated in parallel. A selectall allows two cases to evaluate to true. If two cases are true in a select, incorrect logic may be produced. select generally implies much less logic, however, because there is no implied ordering of the cases.

for statements are used to iterate a single piece of code several times. As a hardware description language, bdsyn must enforce several rules on the for statement. The expressions which specify the start, end, and step values for the loop index must be either constants, or constant expressions. This is because the number of iterations must be fixed. Bdsyn also requires that the index variable of the for statement is a meta-variable. for loops may be nested, and since meta-variables are treated as constants by bdsyn, the inner loop may depend on the outer loop.

```
    FOR i FROM 0 TO 7 DO
        FOR j FROM i TO 7 DO
            statement;
```

The return statement is used to break out of a subroutine and return to the calling routine. If the subroutine returns a value (as described below) the return statement must be followed by an expression which is the return value. The leave statement is similar to return in that can be used to break out of a block of statements. When a leave statement is encountered, control will break out of a labeled statement. The leave must occur within the labeled statement to which it points. For example:

```
loop: FOR i FROM 1 TO 10 DO BEGIN
    statement1;
    IF condition THEN LEAVE loop;
    statement2;
END;
```

Notice that by using the leave it is possible to describe a loop whose number of iterations depends on a logic variable (condition in this case). It is only important that the for statement itself have extents which are constants or meta-variables.

## 3.7 Routines and Routine Calls

Bdsyn has no concept of a main routine, per se. At runtime, bdsyn routines are classified as main routines or subroutines depending on whether or not they called by another routine. Bdsyn allows many main routines in the same model, and these routines can be viewed as running in parallel. It is an error in bdsyn for two disjoint

routines (one is not called by the other) to set the same global variable or primary output of the block. Two disjoint routines may call the same subroutine, however. It is illegal to have recursive routine calls.

Subroutines may return a single variable (regular or meta) to the calling routine. A subroutine that returns a value is declared by placing a bit subscript after the routine name in the routine declaration. The bit subscript specifies the width of the return value or if the value is a meta-variable. If the subscript is missing, the routine returns no value. Parameters passed to subroutines must be declared as part of the routine declaration. Parameter passing has the same semantics as assignment statements (described below).

A common error in bdsyn is to have a subroutine which is not called by another routine. This causes bdsyn to treat the subroutine as a main routine.

## 3.8 Assignment Statements

Assignment statements are written in the form:

```
variable = expression
```

They result is that the variable receives the value of the expression. If the width of the variable is wider than the width of the evaluated expression, extra zeros will be added to pad the high bits of the variable. If the expression is wider than the variable, the high bits are truncated. If the variable is a meta-variable the expression may contain only constants and other meta-variables.

It is possible to assign a value to the same variable twice in the same program. Bdsyn will treat this situation in the same manner as any sequential programming language. In particular the following works correctly.

```
    x = 1;   y = x;   x = 0;
    IF x NEQ y THEN is_ok = 1;
```

Section 4 contains further details on multiple assignment.

## 3.9 Expressions

Legal expressions are of the following form. They are listed in order of precedence, and expressions higher on the list are evaluated first. The curly braces in the zxt, oxt, and sxt expressions represent actual characters in the input, not a repeatable phrase.

```
     |   number                                 ! number as
described above
     |   constant                               ! a declared
constant
     |   meta_variable
     |   regular_variable
     |    (expression)                          !  parenthesized
expression
     |   expression<expression>          ! choose a bit
     |    expression<const_expr:const_expr>         !  choose
several bits
     |   expression & expression         ! concatenation
     |     expression SR0  expression          ! shift  right
filling with zeros
     |     expression SR1  expression          ! shift  right
filling with ones
     |   expression SRR expression        ! rotate right

     |     expression  SL0 expression          ! shift  left
filling with zeros
     |     expression  SL1 expression          ! shift  left
filling with ones
     |   expression SLR expression        ! rotate left
     |   expression * expression          ! multiplication
     |   const_expr / const_expr          ! integer division
     |   const_expr MOD const_expr        ! mod operation
     |   expression + expression          ! addition
     |   expression - expression          ! subtraction

     |   expression EQL expression        ! Equal comparison
     |      expression NEQ  expression         ! Not  equal
comparison
```

```
      |     expression LSS  expression          ! Less  than
comparison
      |    expression LEQ  expression       ! Less than  equal
comparison
      |    expression GTR expression          ! Greater  than

comparison
      |    expression GEQ expression          ! Greater  than
equal comparison
      |   BUF expression                   ! null operation
      |   NOT expression                   ! bitwise NOT
      |   expression AND expression        ! bitwise AND
      |   expression NAND expression       ! bitwise NAND
      |   expression OR expression         ! bitwise OR
      |   expression NOR expression        ! bitwise NOR

      |   expression XOR expression        ! bitwise XOR
      |   expression EQV expression        ! bitwise XNOR
      |   WIDTH expression                 ! return number  of
bits
      |    ZXT {WIDTH =  const_expr} expression      !  zero
extend
      |    OXT {WIDTH  = const_expr} expression       !  one
extend
      |    SXT {WIDTH =  const_expr} expression      !  sign
extend
```

In the above "constexpr" represents a constant expression. A constant expression is an expression that contains only numbers, declared constants, and meta-variables. It may not contain regular variables.

The single bit selection operator, <0>, may take an arbitrary expression as the bit selector. When the bit selector is not a constant, a multiplexor is implied. The multiple bit selection operator, <31:0>, requires that both expressions be a constant expression and that the first number is larger than the second. The concatenation operator `&' joins two bit vectors into one. The expression before the `&' becomes the high order bits, the trailing expression becomes the low order bits.

The shift operations shift the first expression by an amount given by the second expression. When the shift amount is a variable expression, a barrel shifter is thelwidth ofothe shiftedshexpression.on Thiss means itanis possible to shift bits off the end of the expression accidentally. Care should be taken to extend the

expression (using zxt) so no bits are lost.

`+', `-', `*', `/', and mod implement integer arithmetic and are supported for constant expressions. In addition, `+', `-', and `*' are supported for variable expressions. `+' and `-' imply a combinational adder in this case, and

`*' implies a full combinational multiplier. The `*' operation should be used sparingly on variable expressions since it implies so much logic. In particular, `*' should not be used where a shift would be sufficient.

The comparison operators evaluate to a single bit 1 if the condition is true and a single bit 0 if the condition is false. If an expression in the comparison is a variable, a comparator is implied. The bitwise boolean operators take two bit vectors and perform the desired operation bit by bit. If one of the expressions is smaller than the other, it will be automatically zero extended. The width operator returns a constant which is the number of bits in the given expression. zxt, oxt, and sxt will add more significant bits to the given expression to make it the given width. It is an error to specify a width that is smaller than the given expression.

## 3.10 Macro Definitions and Required Files

Bdsyn has the facility for defining macro definitions in the input description. Macro definitions may appear anywhere in the input description and have the following format:

```
    MACRO macro_name = macro_body $ENDMACRO
```

Following a macro definition, any use of macroname will be replaced by the macrobody. The macrobody is expanded exactly as typed (including comments). It may span more than one line and may contain uses of other macro definitions.

Bdsyn also allows distinct input files to be included within other files. This is done using the require command. A require may occur anywhere in a bdsyn file and has the syntax:

```
    REQUIRE 'filename.ext';
```

The text of the required file will be inserted in place of the require command.

## 4 Multiple Assignment

As mentioned in the introduction, bdsyn's input descriptions appear very sequential in nature yet are intended to describe combinational logic. We feel that our form of sequential description gives the user a great deal of power and flexibility in specifying logic. The translation of sequential description to combinational logic is accomplished by correctly handling "multiple assignment."

Our definition of multiple assignment is the correct and consistent handling of many sequential assignments to the same variable. For example:

```
    x = default_value;
    IF condition THEN x = new_value;
```

describes a multiplexer which is controlled by "condition." The above example could be rewritten in a more conventional way:

```
    IF condition THEN
        x = new_value
    ELSE
        x = default_value;
```

At first glance, one might think that multiple assignment is not particularly useful. In fact, multiple assignment is the heart of bdsyn's ability to process sequential descriptions. Several examples of this can be found in the "Examples" section at the end of this guide.

## 5 Unspecified Variables

In bdsyn input descriptions, as in any programming language, it is possible to have variables which are not always defined. For instance, in the following code:

```
    IF cond THEN
        x = expr;
```

the variable x, if it has not previously been assigned to, is unspecified when cond is false.

Under normal circumstances bdsyn will assume that when variables are unspecified, they should have the value zero. This assumption is probably not a good one. In some cases, the unspecified variables could be caused by a mistake in the user specification. In other cases the user may not care about a variable value under certain conditions. In this case the user should probably specify DONTCARE conditions as discussed in section 8.

In general, it is very difficult for bdsyn to detect the use of unspecified variables. However, unspecified variables may be detected by using mis. First, bdsyn should be run using the -n flag. This will cause bdsyn to create a group of variables that end with the characters "**". (Also, the ".inputs" line in the blif output will not be printed.) The blif file containing the "**" variables should then be read into mis and minimized. After minimization, if any of the "**" variables fan out to any gates, this corresponds to the use of an unspecified variable.

## 6 Meta-variables

Meta-variables have a very separate use from logic variables. They can be used within constant expressions because they represent constant values. Unlike logic variables, they are guaranteed to always assume the same particular value. The value of a logic variable may change based on the primary inputs to the bdsyn combinational block.

Because bdsyn is able to statically evaluate the value of meta-variables, it is able to perform some simple logic minimization when meta-variables are involved. In the following example, function1 and function2 are complex functions. (i is a meta-variable.)

```
    FOR i FOR 0 TO 5 DO
        IF i MOD 2 EQL 0 THEN
            x = function1(x)
        ELSE
            x = function2(x);
```

After bdsyn minimization this becomes:

```
    x = function1(x);
    x = function2(x);
    x = function1(x);
    x = function2(x);
    x = function1(x);
    x = function2(x);
```

Had the condition in the if statement above depended on a logic variable, no such minimization would have been possible inside of bdsyn. This is because the value of a logic variable is generally unpredictable.

One non-obvious problem with meta-variables is the following. A meta-variable is defined within a branching structure (i.e. if, select, etc.). If the condition of the branching structure depends on a logic variable, the meta-variable is not correctly defined upon exit from that structure. For example:

```
    meta = 3;
    IF x EQL 1 THEN BEGIN
        meta = 4;
        a<0> = w<meta>;       ! This works correctly
    END
    a<1> = w<meta>;           ! This uses meta = 4 even if  (
x NEQ 1 )
```

This problem stems from indirectly trying to define the meta-variable meta in terms of the logic variable x. For similar reasons, meta-variables that are used for loop counts are not necessarily correct upon exit from the loop. For example:

```
    loop: FOR i FROM 1 TO 10 DO BEGIN
        x = function(x);
        IF x EQL 1 THEN LEAVE loop;
    END;
    y = i;                 ! y gets the value 10  regardless
of the function
```

## 7 Complex Operators

Bdsyn supports several operations which can not be implemented by just one or two gates. These include `+', `-', `*', leq, geq, lss, gtr, shifts, and single bit selection when they are operating on non-constants. The operations imply complex structures such as adders, multipliers, comparators, barrel shifters, and multiplexors. The complex operators are implemented through a set of library routines found in the file "bdsyn.lib". The macros in the library are used to create

routines of an appropriate size (a 3 bit adder, or a 4 bit comparator, etc) which are then inserted into the code.

## 8 Don't Care Conditions

It is often the case that it makes no difference what the value of an output is. For example, in a finite state machine with seven states encoded in three state bits, state 7 never occurs:

```
    SELECT state<2:0> FROM
        [0,1,2,3]: output = 0;
        [4,5,6]: output = 1;
    ENDSELECT;
```

Here, the output for state 7 would be unspecified. To resolve this, a default value for output might be added.

```
    output = 0;
    SELECT state<2:0> FROM
        [0,1,2,3]: output = 0;
        [4,5,6]: output = 1;
    ENDSELECT;
```

The problem is that this is unnecessarily restrictive, and does not allow for all of the potential logic simplification. In actuality, it may make no difference whether "output = 0" or "output = 1" in state 7, because state 7 never occurs. For this reason, bdsyn has the special variable DONTCARE, which will allow logic optimization tools to choose the optimal value for output. DONTCARE is used as any other variable:

```
    output = DONT_CARE;
    SELECT state<2:0> FROM
        [0,1,2,3]: output = 0;
        [4,5,6]: output = 1;
    ENDSELECT;
```

Inside of bdsyn, assignments from this variable will automatically be extended to fit the size of the destination. When DONTCARE is used, it will automatically be added to the input port list in bdsyn's blif output. There is no need to declare the DONTCARE variable.

The current version of mis can not process the don't care conditions produced by bdsyn. This is because don't care specifications in multiple level descriptions can be ambiguous in their meaning. Until a uniform method for handling multiple level don't cares is implemented, it is still possible to use espresso to make partial use of the don't care specification. This can be accomplished by the following (in unix):

```
    bdsyn input_descr | mis -c clp -T pla | \
            espresso -Dmapdc | espresso > output.pla
```

The resulting pla can then be read into mis.

If a description has been written that includes a don't care specification and the use of the don't care specification is not desired, the -z option should be used. The -z option maps every use of DONT-CARE to zero. The effect will be that in places where DONTCARE is specified, the result will be forced to the value zero.

## 9 Examples

Figures 1 through 5 contain five examples of various bdsyn descriptions. The first is a generic finite state machine. It merely generates the next state and control signals based on the present state. The next three descriptions implement the exact same piece of logic. They are included to demonstrate the flexibility

of bdsyn. Note that the second implementation (COUNT2) uses intermediate variables seenZero and seenTrailing as state holders. The variables do not really imply latches to hold the state. They are simply a convenient way to describe an idea.

The fifth example is a complicated decoder circuit. Particular attention should be given to the last statement of bdsyn.in TheudecoderItisidescribedsasn ambarrelt shifter which shifts a single bit 1 onto the correct word line. This is obviously a very inefficient way to build a decoder. It is important to note, however, that bdsyn and mis are capable of reducing the hardware to a more sensible implementation. There is generally no penalty associated with a quick and dirty shortcut description. The logic that is generated is the same.

## 10 Acknowledgments

The parser, inline routine expansion, and macro expansion inside of bdsyn were implemented by Richard Rudell. The for loops, return and leave processing, multiple assignment, and mapping to blif were implemented by Russell Segal. To do logic collapsing, bdsyn uses mis, which was implemented by Albert Wang and Richard Rudell. We would like to thank the students and industrial visitors who participated in the Spring 1986 synthesis class for running bdsyn through its paces. We also would like to thank Professors Richard Newton, Alberto Sangiovanni-Vincentelli, and Carlo Sequin for organizing the synthesis project and given bdsyn a reason to exist.

```
! The classic traffic light controller finite state
machine from
! Mead and Conway, Introduction to VLSI Technology, page
85
MODEL traffic_light
    hl<1:0>,               ! control for highway light
    fl<1:0>,               ! control for farm light
    st<0>,                 ! to start the interval timer
    nextState<1:0> =
    c<0>,                     ! indicating a car on the farm
road
    ts<0>,                 ! timeout of short interval timer
    tl<0>,                 ! timeout of long interval timer
    presentState<1:0> ;
! state assignments
CONSTANT HG = 0, HY = 2, FG = 3, FY = 1;
! symbolic output assignments
CONSTANT GREEN = 0, RED = 1, YELLOW = 2;
ROUTINE traffic_light_controller;

    ! set up default outputs (use of multiple assignment)
    nextState = presentState;
    st = 0;
    SELECT presentState FROM

        [HG]: BEGIN
            hl = GREEN;   fl = RED;
            IF c AND tl THEN BEGIN
                nextState = HY;
                st = 1;
            END;
        [HY]:;BEGIN
            hl = YELLOW;   fl = RED;
            IF ts THEN BEGIN
                nextState = FG;
                st = 1;
            END;
          END;
        [FG]: BEGIN
            hl = RED;   fl = GREEN;
            IF NOT c or tl THEN BEGIN
                nextState = FY;
                st = 1;
            END;
        [FY]:;BEGIN
            hl = RED;   fl = YELLOW;
            IF ts THEN BEGIN
                nextState = HG;
                st = 1;
            END;
          END;
    ENDSELECT;
ENDROUTINE;
ENDMODEL;
```

Figure 1: Traffic Light Controller

```
! COUNT1 (Count Zeros):

! The input is an  8 bit binary string expected to  consist
of
! a string of  1's, a string of 0's,  and a string of  1's.
Any
! of these strings may be zero length.
! For example, `11000111',  `10000001', and `11110000'  are
all
! valid strings, and `10001100' is a mal-formed string.

! The problem  is to design  a circuit which,  given the  8
bit
! string, returns the count of number of consecutive  zeros
in
! the string, or returns  an error condition if the  string
is
! mal-formed.  The output is undefined if the input  string
is
! mal-formed.
MODEL count1 error<0>, out<3:0> = in<7:0>;

! find first bit matching 'val' (return index of the bit)
ROUTINE ff<3:0>(x<7:0>, val<0>);
    STATE i<>;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL val THEN
            RETURN i;
    RETURN 8;
ENDROUTINE;

ROUTINE main;
    STATE x<7:0>;
    ! Shift off the first string of 1's

    x = in SR1 ff(in, 0);
    ! Count is where the first 1 is
    out = ff(x, 1);

    ! Check for a mal-formed string:
    ! shift off 0's, and check for any more 0's
    x = x SR1 out;
    error = ff(x, 0) NEQ 8;
    IF error THEN
        out = DONT_CARE;

ENDROUTINE;
ENDMODEL;
```

Figure 2: Count1

```
! COUNT2 (Count Zeros)
! A different implementation for the last example.

MODEL count2 error<0>, out<3:0> = in<7:0>;
CONSTANT TRUE = 1, FALSE = 0;

ROUTINE legal<0>(x<7:0>);
    STATE i<>;
    STATE seenZero;          ! there has been a zero
     STATE seenTrailing;       ! we  have hit the  trailing
ones field
    seenZero = FALSE;
    seenTrailing = FALSE;
    FOR i FROM 0 TO 7 DO
        IF seenTrailing AND (x<i> EQL 0) THEN
            RETURN FALSE                        !  Illegal
string
        ELSE IF seenZero AND (x<i> EQL 1) THEN
            seenTrailing = TRUE
        ELSE IF x<i> EQL 0 THEN
            seenZero = TRUE;

    ! If we made it to here then the input is legal
    RETURN TRUE;
ENDROUTINE;

ROUTINE zeros<3:0>(x<7:0>);
    STATE i<>, count<3:0>;
    count = 0;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL 0 THEN
            count = count + 1;
    RETURN count;
ENDROUTINE;
ROUTINE main;
    IF legal(in) THEN BEGIN
        error = FALSE;
        out = zeros(in);
    END ELSE BEGIN
        error = TRUE;
        out = DONT_CARE;
    END;
ENDROUTINE;
ENDMODEL;
```

Figure 3: Count2

```
! COUNT3 (Count Zeros)
! Yet another implementation
MODEL count3 error<0>, out<3:0> = in<7:0>;
CONSTANT TRUE = 1, FALSE = 0;

! check for a legal string
ROUTINE legal<0>(x<7:0>);
    STATE i<>, j<>, k<>;

    ! Look for a ...0...1...0...
    FOR i FROM 0 to 5 DO
        FOR j FROM i+1 to 6 DO
            FOR k FROM j+1 to 7 DO
                IF (x<i> EQL 0) AND (x<j> EQL 1) AND
                                    (x<k> EQL 0) THEN
                    ! The input is illegal
                    RETURN FALSE;

    ! If we made it to here then the input is legal
    RETURN TRUE;
ENDROUTINE;


ROUTINE zeros<3:0>(x<7:0>);
    STATE i<>, count<3:0>;
    count = 0;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL 0 THEN
            count = count + 1;
    RETURN count;
ENDROUTINE;
ROUTINE main;
    IF legal(in) THEN BEGIN
        error = FALSE;
        out = zeros(in);
    END ELSE BEGIN
        error = TRUE;
        out = DONT_CARE;
    END;
ENDROUTINE;
ENDMODEL;
```

Figure 4: Count3

```
! Register decoder for the SPUR cpu chip.
! The register file of the SPUR cpu uses a Berkeley RISC
! type register windowing scheme.  The register file is
! divided into 8 overlapping register "windows".  Each
! register window can access 32 of SPUR's 138 registers.
! The first 10 registers of all of the windows point to
! the same 10 "global" registers.  The next 6 registers
! are shared with the top six registers of the previous
!  window.    The  6  top  registers  of  each  window  are
similarly
! shared with the next window.  The last window wraps
! around and shares registers with the first window.
MODEL reg_decode
    addr<137:0>      ! one-hot decoded output (word lines)
=
    cwp<2:0>,        ! Current window pointer
    reg<4:0>;        ! Current register in the window

CONSTANT
    NUMREGS = 138,       ! total number of registers
    NUMGLOBALS = 10;     ! number of global registers

ROUTINE REG_DECODE();
    STATE
        sum<7:0>,        ! temporary variable
         decAddr<7:0>;    ! decoded address (number of  the
word line)

    ! calculate the correct register number
    IF (reg LSS NUMGLOBALS) THEN
        ! point to the global registers
        decAddr = regSpec

    ELSE BEGIN
        sum = (cwp & 0000#2) + regSpec;
         ! Check for wrap around.  If so, need to  subtract
an offset
        IF (sum GTR (NUMREGS - 1)) THEN
            sum = sum - (NUMREGS - NUMGLOBALS);

        decAddr = sum;
    END;

      !  Do the  actual  decoding (***  See main  text  for
details ***)
    addr = (ZXT {width=138} 1) SL0 decAddr;

ENDROUTINE;
ENDMODEL;
```

Figure 5: Register file decoder

D

```
Scripts msu2nwell, msu2pwell, msu2twinwell,
         nwell.fabprep, nwell.fabprep.2u
         pwell.fabprep, pwell.fabprep.2u
         twinwell.fabprep, twinwell.fabprep.2u
         nwell.drc, pwell.drc and twinwell.drc
are Copyright (c) 1990 Massachusetts Technology Park Corp.

Modifications for Octtools release 5.1 are
Copyright (c) 1991 Advanced Hardware Architectures, Inc.

Most of the above scripts are affected. The following scripts, which
are new in Octtools 5.1, are
Copyright (c) 1991 Advanced Hardware Architectures, Inc.

      nwell.fabprep.1.5u,
      pwell.fabprep.1.5u,
      twinwell.fabprep.1.5u

      pla2nwell, pla2pwell, pla2twinwell.

For further information on the use of these scripts for creating CIF
files suitable for fabrication, contact:

      Paul B. Cohen
      Advanced Hardware Architectures, Inc.
      (208) 883-8000
      paul@aha.com
```

---

In order to submit Oct designs for fabrication, there are a few housekeeping details that need to be addressed. One of these is conversion from an Oct database to a properly scaled CIF file.

The layers required to submit a design for fabrication differ slightly from the layers used in Oct. For example, Oct layers NDIF and PDIF are merged to form MOSIS layer CAA (active). In order to distinguish which active areas are N-type and which are P-type, 2 layers are created. These layers are CSN (N-select) and CSP (P-select). CONS, COND, COPS and COPD are combined onto a single MOSIS layer CCA. Some layers only need a name change, such as MET1 becoming CMF.

The best way to handle the creation of a CIF file suitable for fabrication is fabprep (1OCTTOOLS). In order to assure correct handling of geometries near leaf cell boudaries, fabprep first flattens the design to the Oct PHYSICAL level (no instances).

For a large design, the flat database created by fabprep may become quite large. As a result, fabprep may not run to completion due to memory limitations. If this happens, then there are 3 scripts which can be used with octtocif (1OCTTOOLS) to generate a CIF file and take care of the layer re-naming, combining, and generation. While octtocif operates on each cell individually, and therefore may not generate the layers correctly near the cell boundaries, the problems are generally not fatal. (Both fabprep and octtocif have been used to generate CIF files which resulted in working chips.)

If a hierarchical design is read in from a CIF file, and a new CIF file (after modifications?) is written, fabprep will not generate the selects and wells correctly. This is because octflatten will attempt to flatten the Oct database until there are no instances, but the Oct database resulting from using ciftooct does not have any instances. Thus, if an Oct database was created via ciftooct, then octtocif should be used to re-write a CIF file.

Thus, if you can't use fabprep due to design size and/or memory limitations, you can use octtocif with msu2nwell, msu2pwell, or msu2twinwell as conversion files (-f option). If you are using PLAs created with octpla in your design, you should use pla2nwell, pla2pwell, or pla2twinwell instead of msu2[whatever]. It is safe to use the pla scripts with MSU cells. (Actually, for those that have used them before, the pla scripts are actually the OLD MSU scripts. Since the well overlaps were fixed in MSU library version 2.2, the code to fix the wells at the end of the rows was removed. It is still needed for the PLAs, though.)

The scripts (both for fabprep and octtocif) make some assumptions about the layers present in the database. The msu2nwell and pla2nwell scripts assume there is an NWEL layer with geometries, since it uses this information to determine which active areas are source/drain areas and which are well or substrate plugs. Msu2pwell and pla2pwell use the PWEL layer for determining the same information.

# E

```
static char rcs_id[] = "$Id: attach.c,v 1.7 90/04/27 09:28:24 octtools Exp $";
#include "copyright.h"
#include "port.h"
#include "internal.h"

octStatus octUnDetach(container, object)
struct octObject *container;
struct octObject *object;
{
    generic *cptr, *optr;
    octStatus retval;

    cptr = oct_id_to_ptr(container->objectId);
    if (cptr == NIL(generic)) {
      oct_error("octUnDetach: The first argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    optr = oct_id_to_ptr(object->objectId);
    if (optr == NIL(generic)) {
      oct_error("octUnDetach: The second argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    retval = oct_do_undetach(cptr, optr);
    if (retval < OCT_OK) {
      oct_prepend_error("octUnDetach: ");
    }
    return retval;
}

oct_do_undetach(cptr, optr)
generic *cptr;
generic *optr;
{
    int retval;
    struct facet *desc;

    retval = (*oct_object_descs[cptr->flags.type].undetach_func)(cptr, optr);

    if (retval < OCT_OK || retval == OCT_ALREADY_ATTACHED) {
      return retval;
    }

    desc = cptr->facet;
    desc->attach_date = 輸>time_stamp;

    RECORD_ATTACH(cptr, optr, retval);
    return retval;
}

octStatus octAttach(container, object)
struct octObject *container;
struct octObject *object;
{
    generic *cptr, *optr;
    octStatus retval;

    cptr = oct_id_to_ptr(container->objectId);
    if (cptr == NIL(generic)) {
      oct_error("octAttach: The first argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    optr = oct_id_to_ptr(object->objectId);
    if (optr == NIL(generic)) {
      oct_error("octAttach: The second argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    retval = oct_do_attach(cptr, optr, 0);
    if (retval < OCT_OK) {
      oct_prepend_error("octAttach: ");
    }
    return retval;
}

oct_do_attach(cptr, optr, only_once)
generic *cptr;
generic *optr;
int only_once;
{
    int retval;
    struct facet *desc;

    if (cptr->facet != optr->facet) {
      oct_error("can not attach across facets (yet)");
      errRaise(OCT_PKG_NAME, OCT_ERROR, octErrorString());
    }

    retval = (*oct_object_descs[cptr->flags.type].attach_func)(cptr, optr,
only_once);

    if (retval < OCT_OK || retval == OCT_ALREADY_ATTACHED) {
      return retval;
    }

    desc = cptr->facet;
    desc->attach_date = 輸>time_stamp;

    RECORD_ATTACH(cptr, optr, retval);
    return retval;
}

octStatus octAttachOnce(container, object)
struct octObject *container;
struct octObject *object;
{
    generic *cptr, *optr;
    octStatus retval;

    cptr = oct_id_to_ptr(container->objectId);
    if (cptr == NIL(generic)) {
      oct_error("octAttachOnce: The first argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    optr = oct_id_to_ptr(object->objectId);
    if (optr == NIL(generic)) {
      oct_error("octAttachOnce: The second argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    retval = oct_do_attach(cptr, optr, 1);

    if (retval < OCT_OK) {
      oct_prepend_error("octAttachOnce: ");
    }
    return retval;
}

octStatus
octUnattach(container, object)
struct octObject *container;
struct octObject *object;
{
    generic *cptr, *optr;
    octStatus retval;
      char buffer[200];

    cptr = oct_id_to_ptr(container->objectId);
    if (cptr == NIL(generic)) {
      sprintf(buffer,"octUnAttach: The first argument's id (%d) has been
corrupted", container->objectId);
      oct_error(buffer);
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    optr = oct_id_to_ptr(object->objectId);
    if (optr == NIL(generic)) {
      sprintf(buffer,"octUnAttach: The second argument's id (%d) has been
corrupted", object->objectId);
      oct_error(buffer);
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    retval = oct_do_unattach(cptr, optr);
    if (retval < OCT_OK) {
      oct_prepend_error("octUnAttach: ");
    }
    return retval;
}

oct_do_unattach(cptr, optr)
generic *cptr;
generic *optr;
{
    struct facet *desc;
    int retval, rc_retval;
    retval = (*oct_object_descs[cptr->flags.type].unattach_func)(cptr, optr);

    if (retval < OCT_OK) {
      return retval;
    }

    desc = cptr->facet;
    desc->detach_date = 輸>time_stamp;

    RECORD_DETACH(cptr, optr, rc_retval);

    if (rc_retval < OCT_OK) {
      return rc_retval;
    } else {
      return retval;
    }
}

octStatus
octDetach(container, object)
struct octObject *container;
struct octObject *object;
{
    generic *cptr, *optr;
    octStatus retval;
      char buffer[200];

    cptr = oct_id_to_ptr(container->objectId);
    if (cptr == NIL(generic)) {
      sprintf(buffer,"octDetach: The first argument's id (%d) has been
corrupted", container->objectId);
      oct_error(buffer);
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    optr = oct_id_to_ptr(object->objectId);
    if (optr == NIL(generic)) {
      sprintf(buffer,"octDetach: The second argument's id (%d) has been
corrupted", object->objectId);
      oct_error(buffer);
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    retval = oct_do_detach(cptr, optr);
    if (retval < OCT_OK) {
      oct_prepend_error("octDetach: ");
    }
    return retval;
}

oct_do_detach(cptr, optr)
generic *cptr;
generic *optr;
{
    struct facet *desc;
    int retval, rc_retval;

    retval = (*oct_object_descs[cptr->flags.type].detach_func)(cptr, optr);

    if (retval < OCT_OK) {
      return retval;
    }

    desc = cptr->facet;
    desc->detach_date = 輸>time_stamp;

    RECORD_DETACH(cptr, optr, rc_retval);

    if (rc_retval < OCT_OK) {
      return rc_retval;
    } else {
      return retval;
    }
}


octStatus
oct_unattach_default(cptr, optr)
generic *cptr;
generic *optr;
{
    return oct_detach_content(cptr, optr);
}


octStatus
oct_undetach_default(cptr, optr)
generic *cptr;
generic *optr;
{
    int ctype = cptr->flags.type;
    int otype = optr->flags.type;

    return oct_attach_content(cptr, optr, 0);
}


octStatus
oct_detach_default(cptr, optr)
generic *cptr;
generic *optr;
{
    return oct_detach_content(cptr, optr);
}


octStatus
oct_attach_default(cptr, optr, only_once)
generic *cptr;
generic *optr;
int only_once;
```

```
{
    int ctype = cptr->flags.type;
    int otype = optr->flags.type;

    if (!IN_MASK(otype, oct_object_descs[ctype].contain_mask)) {
      oct_error("Objects of type %s cannot contain objects of type %s",
                oct_type_names[ctype], oct_type_names[otype]);
      return OCT_ERROR;
    }
    return oct_attach_content(cptr, optr, only_once);
}

octStatus octIsAttached(container, object)
struct octObject *container;
struct octObject *object;
{
    generic *cptr, *optr;
    octStatus retval;

    cptr = oct_id_to_ptr(container->objectId);
    if (cptr == NIL(generic)) {
      oct_error("octIsAttached: The first argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    optr = oct_id_to_ptr(object->objectId);
    if (optr == NIL(generic)) {
      oct_error("octIsAttached: The second argument's id has been corrupted");
      errRaise(OCT_PKG_NAME, OCT_CORRUPTED_OBJECT, octErrorString());
    }

    if (cptr->facet != optr->facet) {
      return OCT_NOT_ATTACHED;
    }

    retval = (*oct_object_descs[cptr->flags.type].is_attached_func)(cptr, optr);

    if (retval < OCT_OK) {
      oct_prepend_error("octIsAttached: ");
    }
    return retval;
}

octStatus
oct_is_attached_default(cptr, optr)
generic *cptr;
generic *optr;
{
    struct chain *ptr;

    ptr = optr->containers;
    for(;ptr != NIL(struct chain); ptr = ptr->next_chain) {
      if (ptr->container == cptr) {
          return OCT_OK;
      }
    }
    return OCT_NOT_ATTACHED;
```

# F

```
#ifndef OCT_H
#define OCT_H

#define OCT_LEVEL 2

#ifndef OCT_1_NONCOMPAT
#ifndef OCT_1_COMPAT
#define OCT_1_COMPAT
#endif /* OCT_1_COMPAT */
#endif /* OCT_1_NONCOMPAT */

#ifndef PORT_H
#include "port.h"
#endif /* PORT_H */

#ifndef ANSI_H
#include "ansi.h"
#endif /* ANSI_H */

#ifndef FILE
#include <stdio.h>            /* Because some functions require FILE* args. */
#endif

#define OCT_PKG_NAME "oct"

typedef int32 octCoord;
#define OCT_MAX_COORD LONG_MAX
#define OCT_MIN_COORD -(OCT_MAX_COORD)

#ifdef BIGID
typedef struct {
      int32 idmsb;
      int32 idlsb;
} octId;

extern octId oct_null_id;
#else
typedef int32 octId;
#define oct_null_id ((octId) 0)
#endif
#define OCT_NULL_ID oct_null_id

#define OCT_CURRENT_VERSION ""

#define OCT_INCONSISTENT_BAG "*OCT_INCONSISTENT*"
#define OCT_CHILD 0x01
#define OCT_SIBLING     0x02

#define octGenerate(generator,object) \
     (*(generator)->generator_func)((generator),(object))

     /* to be expanded as needed */

typedef int octStatus;
#define OCT_INCONSISTENT 8
#define OCT_ALREADY_ATTACHED 7
#define OCT_NOT_ATTACHED 6
```

```
#define OCT_ALREADY_OPEN 5
#define OCT_GEN_DONE  4
#define OCT_OLD_FACET 3
#define OCT_NEW_FACET 2
#define OCT_OK 1
#define OCT_ERROR -1
#define OCT_NO_EXIST -2
#define OCT_NO_PERM -3
#define OCT_CORRUPTED_OBJECT -4           /* errRaise - done */
#define OCT_NOT_FOUND -5
#define OCT_TOO_SMALL -6
#define OCT_ILL_OP -7               /* errRaise - done */
#define OCT_NO_BB -8

/* modes for octOpenFacet */

#define OCT_READ 1
#define OCT_OVER_WRITE 2
#define OCT_APPEND 3
#define OCT_REVERT 4

enum octPropType {
    OCT_NULL=0, OCT_INTEGER, OCT_REAL, OCT_STRING, OCT_ID, OCT_STRANGER,
    OCT_REAL_ARRAY, OCT_INTEGER_ARRAY
};
typedef enum octPropType octPropType;

struct octUserValue {
    int32 length;
    char *contents;
};

struct octProp {
    char *name;
    octPropType type;
    union octValue {
      int32 integer;
      double real;
      char *string;
      octId id;
      struct {
          int32 length;
          int32 *array;
      } integer_array;
      struct {
          int32 length;
          double *array;
      } real_array;
      struct octUserValue stranger;
    } value;
};

struct octPoint {
    octCoord x;
    octCoord y;
};
```

```
struct octEdge {
    struct octPoint start;
    struct octPoint end;
};

struct octBox {
    struct octPoint lowerLeft;
    struct octPoint upperRight;
};

struct octPolygon {
    int unused;     /* So that it is not an empty structure. */
};

struct octCircle {
    octCoord startingAngle;
    octCoord endingAngle;
    octCoord innerRadius;
    octCoord outerRadius;
    struct octPoint center;
};

struct octPath {
    octCoord width;
};

/* for vertical justification */
#define OCT_JUST_BOTTOM 0
#define OCT_JUST_CENTER 1
#define OCT_JUST_TOP    2

/* for horizontal justification -- also uses OCT_JUST_CENTER */
#define OCT_JUST_LEFT   0
#define OCT_JUST_RIGHT  2

struct octLabel {
    char *label;
    struct octBox region;
    octCoord textHeight;
    unsigned vertJust : 2;    /* vert position of text block in region */
    unsigned horizJust : 2;   /* horiz position of text block in region */
    unsigned lineJust : 2;    /* horiz position of each line in text block */
};

enum octTransformType {
    OCT_NO_TRANSFORM = 0,
    OCT_MIRROR_X = 1,
    OCT_MIRROR_Y = 2,
    OCT_ROT90 = 3,
    OCT_ROT180 = 4,
    OCT_ROT270 = 5,
    OCT_MX_ROT90 = 6,
    OCT_MY_ROT90 = 7,
    OCT_FULL_TRANSFORM = 8
};

typedef enum octTransformType octTransformType;
```

```
struct octTransform {
    struct octPoint translation;
    octTransformType transformType;
    double generalTransform[2][2];
};

struct octInstance {
    char *name;
    char *master;
    char *view;
    char *facet;
    char *version;
    struct octTransform transform;
};

struct octNet {
    char *name;
    int32 width;
};

struct octTerm  {
    char *name;
    octId instanceId;
    int32 formalExternalId;
    int32 width;
};

struct octBag {
    char *name;
};

struct octLayer {
    char *name;
};

typedef int32 octObjectMask;
typedef int32 octFunctionMask;

struct octChangeList {
    octObjectMask objectMask;
    octFunctionMask functionMask;
};

struct octChangeRecord {
    int32 changeType;
    int32 objectExternalId;
    int32 contentsExternalId;
};

#define OCT_UNDEFINED_OBJECT 0
#define OCT_FACET 1
#define OCT_TERM 2
#define OCT_NET 3
#define OCT_INSTANCE 4
#define OCT_POLYGON 5
#define OCT_BOX 6
```

```
#define OCT_CIRCLE 7
#define OCT_PATH 8
#define OCT_LABEL 9
#define OCT_PROP 10
#define OCT_BAG 11
#define OCT_LAYER 12
#define OCT_POINT 13
#define OCT_EDGE 14
#define OCT_FORMAL 15
#define OCT_MASTER 16
#define OCT_CHANGE_LIST 17
#define OCT_CHANGE_RECORD 18
#define OCT_MAX_TYPE 18

#define OCT_FACET_MASK ((octObjectMask) (1<<OCT_FACET))
#define OCT_TERM_MASK ((octObjectMask) (1<<OCT_TERM))
#define OCT_NET_MASK ((octObjectMask) (1<<OCT_NET))
#define OCT_INSTANCE_MASK ((octObjectMask) (1<<OCT_INSTANCE))
#define OCT_PROP_MASK ((octObjectMask) (1<<OCT_PROP))
#define OCT_BAG_MASK ((octObjectMask) (1<<OCT_BAG))
#define OCT_POLYGON_MASK ((octObjectMask) (1<<OCT_POLYGON))
#define OCT_BOX_MASK ((octObjectMask) (1<<OCT_BOX))
#define OCT_CIRCLE_MASK ((octObjectMask) (1<<OCT_CIRCLE))
#define OCT_PATH_MASK ((octObjectMask) (1<<OCT_PATH))
#define OCT_LABEL_MASK ((octObjectMask) (1<<OCT_LABEL))
#define OCT_LAYER_MASK ((octObjectMask) (1<<OCT_LAYER))
#define OCT_POINT_MASK ((octObjectMask) (1<<OCT_POINT))
#define OCT_EDGE_MASK ((octObjectMask) (1<<OCT_EDGE))
#define OCT_FORMAL_MASK ((octObjectMask) (1<<OCT_FORMAL))
#define OCT_CHANGE_LIST_MASK ((octObjectMask) (1<<OCT_CHANGE_LIST))
#define OCT_CHANGE_RECORD_MASK ((octObjectMask) (1<<OCT_CHANGE_RECORD))

#define OCT_MARK -1
#define OCT_CREATE 1
#define OCT_DELETE 2
#define OCT_MODIFY 3
#define OCT_PUT_POINTS 4
#define OCT_ATTACH_CONTENT 5
#define OCT_ATTACH_CONTAINER 6
#define OCT_DETACH_CONTENT 7
#define OCT_DETACH_CONTAINER 8
#define OCT_MARKER 9

#define OCT_CREATE_MASK ((octFunctionMask) (1<<OCT_CREATE))
#define OCT_DELETE_MASK ((octFunctionMask) (1<<OCT_DELETE))
#define OCT_MODIFY_MASK ((octFunctionMask) (1<<OCT_MODIFY))
#define OCT_PUT_POINTS_MASK ((octFunctionMask) (1<<OCT_PUT_POINTS))
#define OCT_DETACH_CONTENT_MASK ((octFunctionMask) (1<<OCT_DETACH_CONTENT))
#define OCT_DETACH_CONTAINER_MASK ((octFunctionMask) (1<<OCT_DETACH_CONTAINER))
#define OCT_ATTACH_CONTENT_MASK ((octFunctionMask) (1<<OCT_ATTACH_CONTENT))
#define OCT_ATTACH_CONTAINER_MASK ((octFunctionMask) (1<<OCT_ATTACH_CONTAINER))
#define OCT_ATTACH_MASK
((octFunctionMask)(OCT_ATTACH_CONTENT_MASK|OCT_ATTACH_CONTAINER_MASK))
#define OCT_DETACH_MASK
((octFunctionMask)(OCT_DETACH_CONTENT_MASK|OCT_DETACH_CONTAINER_MASK))
#define OCT_MARKER_MASK ((octFunctionMask) (1<<OCT_MARKER))
#define OCT_ALL_FUNCTION_MASK ((octFunctionMask)\
(OCT_CREATE_MASK|OCT_DELETE_MASK|OCT_MODIFY_MASK|OCT_ATTACH_MASK|OCT_DETACH_MASK
|OCT_PUT_POINTS_MASK))

#define OCT_GEO_MASK ((octObjectMask)\
  (OCT_POLYGON_MASK|OCT_CIRCLE_MASK|OCT_PATH_MASK|OCT_LABEL_MASK|OCT_BOX_MASK|\
   OCT_POINT_MASK|OCT_EDGE_MASK))

#define OCT_ALL_MASK ((octObjectMask)\
  (OCT_GEO_MASK|OCT_TERM_MASK|OCT_NET_MASK|OCT_INSTANCE_MASK|OCT_PROP_MASK|\

OCT_LAYER_MASK|OCT_BAG_MASK|OCT_FACET_MASK|OCT_FORMAL_MASK|OCT_CHANGE_LIST_MASK|
\
   OCT_CHANGE_RECORD_MASK))

struct octFacet {
    char *cell;
    char *view;
    char *facet;
    char *version;
    char *mode;
};

struct octFacetInfo {
    int32 timeStamp;
    int32 bbDate;
    int32 formalDate;
    int32 createDate;
    int32 deleteDate;
    int32 modifyDate;
    int32 attachDate;
    int32 detachDate;
    int numObjects;
    struct octBox bbox;
    char *fullName;
};

struct octObject {
    int type;
    octId objectId;
    union {
      struct octFacet facet;
      struct octInstance instance;
      struct octProp prop;
      struct octTerm term;
      struct octNet net;
      struct octBox box;
      struct octPolygon polygon;
      struct octCircle circle;
      struct octPath path;
      struct octLabel label;
      struct octBag bag;
      struct octLayer layer;
      struct octPoint point;
      struct octEdge edge;
      struct octChangeRecord changeRecord;
      struct octChangeList changeList;
    } contents;
};


struct octGenerator {
    octStatus (*generator_func)
      ARGS((struct octGenerator *generator, struct octObject *object));
    char *state;
    octStatus (*free_func)
      ARGS((struct octGenerator *generator));
};

typedef struct octGenerator octGenerator;

typedef struct octObject octObject;
typedef struct octBox octBox;
typedef struct octPolygon octPolygon; /* NEW (Aug 7 1991)*/
typedef struct octPoint octPoint;
typedef struct octTransform octTransform;

#ifdef OCT_1_COMPAT
#define octOpenMaster compatOctOpenMaster
#endif

EXTERN void octBegin
      NULLARGS;
EXTERN void octEnd
      NULLARGS;
EXTERN void octError
      ARGS((char *));
EXTERN char *octErrorString
      NULLARGS;

EXTERN octStatus octBB
      ARGS((octObject *object, octBox *box));
EXTERN octStatus octOpenFacet
      ARGS((octObject *facet));
EXTERN octStatus octOpenMaster
      ARGS((octObject *instance, octObject *master));
EXTERN octStatus octWriteFacet
      ARGS((octObject *newf, octObject *old));
EXTERN octStatus octCopyFacet
      ARGS((octObject *newf, octObject *old));
EXTERN octStatus octCloseFacet
      ARGS((octObject *facet));
EXTERN octStatus octFlushFacet
      ARGS((octObject *facet));
EXTERN octStatus octFreeFacet
      ARGS((octObject *facet));
EXTERN octStatus octCreateOrModify
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octGetById
      ARGS((octObject *object));
EXTERN octStatus octGetByName
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octGetContainerByName
      ARGS((octObject *container, octObject *object));
```

```
EXTERN void octGetFacet
      ARGS((octObject *object, octObject *facet));
EXTERN octStatus octGetOrCreate
      ARGS((octObject *container, octObject *object));

EXTERN octStatus octGetPoints
      ARGS((octObject *object, int32 *num_points, struct octPoint *points));
EXTERN octStatus octPutPoints
      ARGS((octObject *object, int32 num_points, struct octPoint *points));

EXTERN octStatus octScaleAndModifyGeo
      ARGS((octObject *object, double scale));
EXTERN void octScaleGeo
      ARGS((octObject *object, double scale));
EXTERN octStatus octTransformAndModifyGeo
      ARGS((octObject *object, struct octTransform *transform));
EXTERN void octTransformGeo
      ARGS((octObject *object, struct octTransform *transform));
EXTERN void octTransformPoints
      ARGS((int num_points, struct octPoint *points, struct octTransform
*transform));

EXTERN octStatus octPrintObject
      ARGS((FILE *outfile, octObject *object, int));

EXTERN octStatus octAttach
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octAttachOnce
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octDetach
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octIsAttached
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octCreate
      ARGS((octObject *container, octObject *object));
EXTERN octStatus octDelete
      ARGS((octObject *object));
EXTERN octStatus octModify
      ARGS((octObject *object));

EXTERN octStatus octGenFirstContainer
      ARGS((octObject *object, octObjectMask mask, octObject *container));
EXTERN octStatus octGenFirstContent
      ARGS((octObject *container, octObjectMask mask, octObject *object));
EXTERN octStatus octInitGenContainers
      ARGS((octObject *object, octObjectMask mask, octGenerator *gen));
EXTERN octStatus octInitGenContents
      ARGS((octObject *object, octObjectMask mask, octGenerator *gen));
EXTERN octStatus octInitGenContentsWithOffset
      ARGS((octObject *obj, octObjectMask mask, octGenerator *gen, octObject
*offset));
EXTERN octStatus octFreeGenerator
      ARGS((octGenerator *gen));

EXTERN octStatus octGetByExternalId
      ARGS((octObject *container, int32 xid, octObject *object));
EXTERN void octExternalId
```

```
      ARGS((octObject *object, int32 *xid));
EXTERN octStatus octInitUserGen
        ARGS((char *user_state,octStatus (*gen_func)(),octStatus
(*free_func)(),octGenerator *gen));
EXTERN void octWriteStats
      ARGS((octObject *object, FILE *file));
EXTERN octStatus octGetFacetInfo
      ARGS((octObject *object, struct octFacetInfo *info));
EXTERN int octIsTemporary
      ARGS((octObject *object));
EXTERN void octMarkTemporary
      ARGS((octObject *object));
EXTERN void octUnmarkTemporary
      ARGS((octObject *object));
EXTERN octStatus octOpenRelative
      ARGS((octObject *robj, octObject *fobj, int location));
EXTERN void octFullName
      ARGS((octObject *facet, char **name));

EXTERN int octIdsEqual
      ARGS((octId id1, octId id2));
EXTERN int octIdIsNull
      ARGS((octId id));
EXTERN char *octFormatId
      ARGS((octId id));

/* for use with the `st' package */

EXTERN int octIdCmp
      ARGS((const char *id1, const char *id2));
EXTERN int octIdHash
      ARGS((char *id1, int mod));

#endif /* OCT_H */
```

# G

-------------- Octtools 5.1 ------------------------------------

Octtools is a collection of programs and libraries that form an integrated system for IC design. The system includes tools for PLA and multiple-level logic synthesis, state assignment, standard-cell, gate-matrix and macro-cell placement and routing, custom-cell design, circuit, switch and logic-level simulation, and a variety of utility programs for manipulating schematic, symbolic, and geometric design data. Most tools are integrated with the Oct data manager and the VEM user interface.

The Octtools distribution includes over 60 programs and 30 libraries, some cell libraries (including the MSU 2.2 Standard Cell Library).

The software comes on a tape with about 32MB of compressed archives.
The unfolding of the distribution requires at least 66 MB, more (about 50MB more) if you
decide to unfold optional parts such as Spice, OctLisp, Bear-FP, or other unsupported code.
About 130MB are required to build all the programs. After that, executable can be stripped manually to save about 20MB. On the IBM RS/6000 the disk requirements are about twice as much.

The software requires UNIX, the window system X11R4 including the Athena Widget Set. The design manager VOV and a few other tools require the C++ compiler \fBg++\fR.

Octtools-5.1 have been built and tested on the following combinations of machines and operating systems: DECstation 3100, 5000 running Ultrix 4.1 and 4.2; DEC VAX running Ultrix 4.1 and 4.2; Sun 3 and 4 running OS 4.0 and Sun SparcStation running OS 4.0. The program has been tried on the following machines, but is not supported: Sequent Symmetry, IBM RS/6000 running AIX 3.1.

To obtain a copy of Octtools 5.1 (8mm, tk50, or 1/4inch cartridge QIC150) and a printed copy of the documentation) for a $250 distribution charge, contact:

Cindy Manly-Fields
Industrial Liaison Program
479 Cory Hall
University of California
Berkeley, CA 94720

# H

REDACTED
IN ITS ENTIRETY

# I

**REDACTED**
**IN ITS ENTIRETY**

# J

**REDACTED**
**IN ITS ENTIRETY**

# K

**REDACTED**
**IN ITS ENTIRETY**